

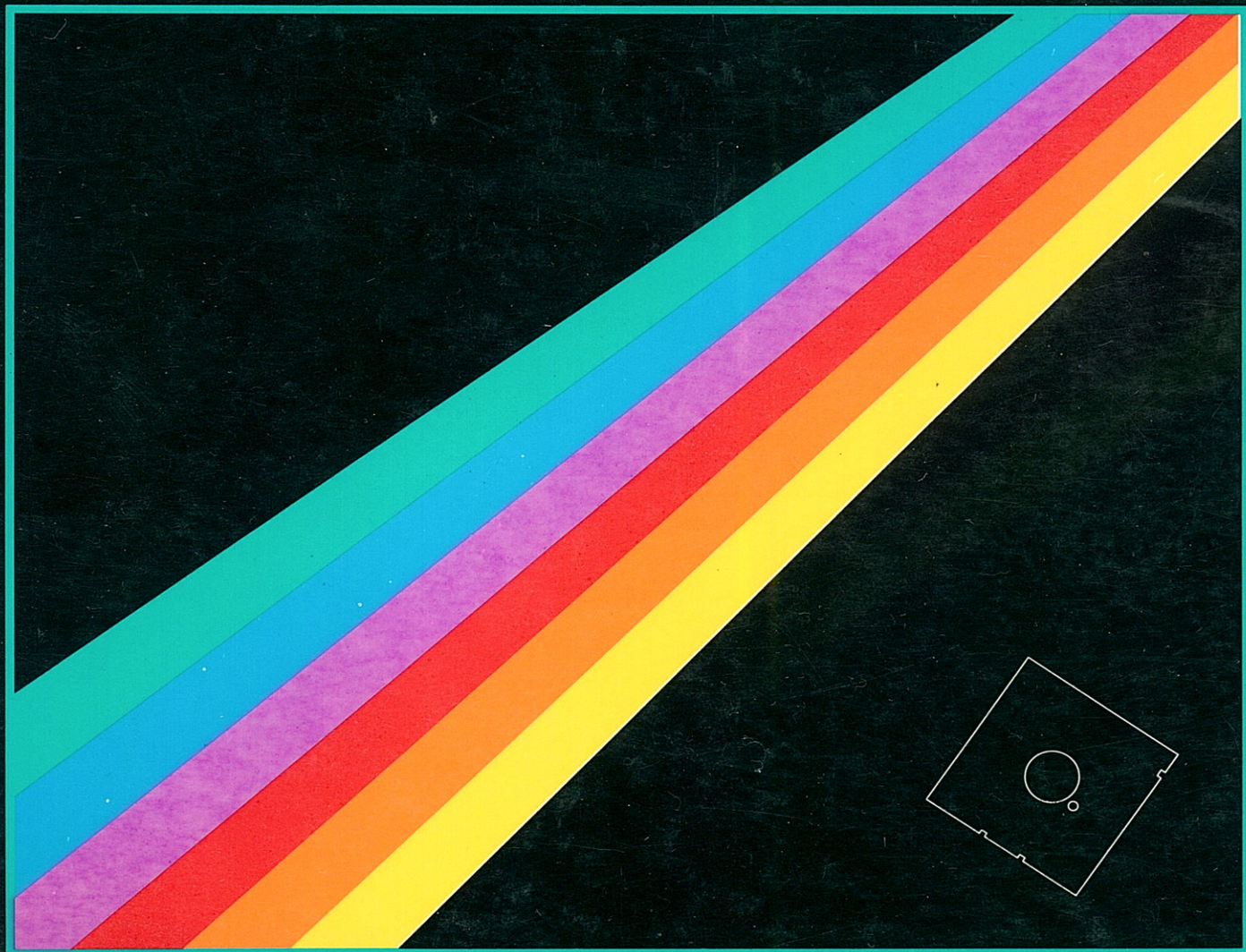
LÁSZLÓ JÓZSEF

A VGA-KÁRTYA PROGRAMOZÁSA

PASCAL ÉS ASSEMBLY NYELVEN



COMPUTERBOOKS



LÁSZLÓ JÓZSEF

A VGA-KÁRTYA PROGRAMOZÁSA PASCAL ÉS ASSEMBLY NYELVEN

LEKTOR
DRÓTOS DÁNIEL



COMPUTERBOOKS
Budapest, 1997

Minden jog fenntartva. Jelen könyvet vagy annak részeit a Kiadó engedélye nélkül bármilyen formában, vagy eszközzel reprodukálni tilos.

©László József, 1995

ISBN: 963 618 154 3

©Kiadó: ComputerBooks Kiadói Kft
1126 Bp. Tartsay V. u 12
☎ 1751-564, fax: 1753-591
E-mail: info@computerbooks.hu
<http://www.computerbooks.hu>
Felelős kiadó: ComputerBooks Kft ügyvezetője
Borítóterv: Székely Edith

Tartalom

Előszó	7
Mire lesz szükség?	9
1. Alapfogalmak	11
1.1. A monitorok elvi működése	11
1.1.1. Monokrom képábrázolási módszer.....	11
1.1.2. Színes képábrázolási módszer	13
1.2. Az RGB színkeverési módszer.....	14
2. Képernyőszerkezetek	19
2.1. A karakteres kép felépítése	19
2.2. A monokrom grafikus kép szerkezete.....	22
2.3. A 16 színű grafikus kép szerkezete	24
2.4. A 256 színű grafikus kép szerkezete	28
2.5. A True Color (valós színű) grafikus kép.....	30
2.6. A karakter bittérképek.....	30
3. A VGA kártya regiszterkészlete	33
3.1. A VGA latch	33
3.2. Az Attribute Controller regiszterek.....	34
3.3. Státusz- és kimeneti regiszterek.....	36
3.4. Video-sequencer regiszterek	37
3.5. RGB regiszterek.....	40
3.6. Grafikai vezérlő regiszterek	41
3.7. Elektronsugár-vezérlő regiszterek (CRTC).....	44
3.8. Egyéb regiszterek.....	48
3.9. A képmemória olvasása és írása.....	49
3.9.1. Olvasási módok	49
3.9.2. Írás módok.....	50
4. A VGA-BIOS szubrutinjai	55
5. A VGA kártya programozása szöveges módban.....	69
5.1. Néhány hasznos szubrutin.....	69
5.2. Felhasználói karakterkészletek.....	76
5.3. A karakterkészlet elérése a memóriában	79
5.4. Karaktermódosítás megszakítás alatt	83
5.5. A karakteres képernyő görgetése	87
5.6. Pixeles görgetés karakteres képernyőn	91
5.7. A billentyűzetmegszakítás működése, saját billentyűkezelés	99
5.8. Osztott képernyő karakteres módban	102
5.9. Labirintus játék	106
5.9.1. A karakterkészlet.....	106
5.9.2. Eljárások és függvények.....	109

5.10. A CrtPlus egység.....	113
5.10.1. Az eljárások és a függvények csoportjai	114
5.10.2. Változók, állandók.....	114
5.10.3. Az eljárások és függvények működése.....	115
5.11. A CharEdit segédprogram.....	121
6. A paletta programozása	123
6.1. Egy kis színelmélet	123
6.2. A színkeverés alapelve.....	125
6.3. A lineáris színkeverés	126
6.4. Árnyalatok szögfüggvények alapján	128
6.5. Ha csak tizenhat színünk van... ..	129
6.6. Animáció és mozgatás a paletta segítségével.....	131
6.7. Axonometrikus ábrázolás és színrotáció	136
6.8. A SetVGAPalette eljárás	140
7. A tizenhat színű grafikus módok programozása.....	141
7.1. Egy pixel kirajzolása és lekérdezése	142
7.1.1. Egy pixel kigyújtása	142
7.1.2. Egy pixel színének lekérdezése	144
7.2. Vonalrajzoló algoritmus.....	146
7.3. A GrPlus egység.....	150
7.3.1. Változók és állandók	151
7.3.2. Eljárások.....	152
7.3.3. Az eljárások működése	155
7.3.4. Az írás mód átváltása.....	155
7.3.5. Karakterek kiírása bájtárra.....	156
7.3.6. Karakterek kiírása pixelpozícióba	159
7.3.7. A képernyő inicializálása.....	163
7.3.8. Különleges vonalrajzoló eljárások.....	163
7.3.9. Pont kirajzolása és lekérdezése.....	168
7.3.10. Téglalapok rajzolása	168
7.4. Grafikus objektum: BOB	172
7.5. A BOB megjelenítésének nehézségei.....	173
7.6. Egyszerű egyszínű megjelenítő.....	175
7.7. A háttér megőrzése: kettősbuffer-technika	180
7.8. Többszínű, nem átfedő BOB-ok	192
7.9. Többszínű, átfedő BOB-ok	203
7.10. Animáció, BOB mozgásfázisok	215
7.11. Grafikus kép görgetése.....	218
7.11.1. Ablakon belüli görgetés.....	218
7.11.2. Osztott képernyő és görgetés (scroll)	221
7.12. Képmemória tárolása és mozgatása	227
7.12.1. Tárolás a heapben.....	227
7.12.2. Tárolás a videomemóriában	232
7.12.3. Tárolás a háttértárolón.....	235

8. A 256 színű grafikus mód programozása I. rész	237
8.1. Pont kirajzolása és lekérdezése	240
8.1.1. Egy pixel kigyújtása	240
8.1.2. Egy képpont színének lekérdezése	241
8.2. Vonalrajzolás	241
8.3. RGB beállítás	245
8.4. Téglalapok rajzolása	245
8.4.1. Kitöltött téglalap	245
8.4.2. Üres téglalap	247
8.4.3. Vastagon keretezett téglalap	248
8.4.4. Vékonyan keretezett téglalap	251
8.5. Karakterek kiírása a 256 színű képre	251
8.5.1. Egy karakter kiírása	253
8.5.2. Egy sztring kiírása	254
8.6. Képterületek tárolása és másolása	255
8.7. Ablakon belüli görgetés	259
8.8. Egyéb eljárások	260
8.9. BOB-ok a 256 színű képen	261
9. A 256 színű grafikus mód programozása II. rész	271
9.1. A módosított bájt/pixel képszerkezet	271
9.2. A képszerkezethez tartozó unit	273
9.3. A videomód beállítása	276
9.4. Pont kirajzolása és lekérdezése	277
9.4.1. Egy pixel kigyújtása	277
9.4.2. Egy pixel színének lekérdezése	279
9.5. Vonalrajzolás	279
9.5.1. Tetszőleges irányú vonal rajzolása	279
9.5.2. Függőleges vonal rajzolása	284
9.5.3. Vízszintes vonal rajzolása	285
9.6. Téglalapok rajzolása	287
9.6.1. Üres téglalap rajzolása	287
9.6.2. Kitöltött téglalap rajzolása	287
9.6.3. Keretezett téglalapok	290
9.7. Képterületek tárolása és másolása	290
9.8. Karakter és szöveg megjelenítése	294
9.9. Paletta beállítása és lekérdezése	298
9.10. A képmemória törlése	299
9.11. BOB-ok megjelenítése a kettősbuffer-módszerrel	300
9.12. Teljes képernyő görgetése	314
10. Nem szabványos videomódok	321
10.1. Szöveges módok	324
10.2. Bitplane szervezésű grafikus módok	327
10.3. A 256 színű módok	329

11. SVGA kártyák, a T8900C SVGA	333
11.1. Trident SVGA videomódok	333
11.2. Szöveges módok	334
11.3. Bitplane szervezésű grafikus módok.....	335
11.4. Bájt/pixel szervezésű grafikus módok.....	335
11.4.1. A bank lapozási technika.....	335
11.4.2. A T8900C egység.....	337
Tárgymutató.....	341
Táblázatok	347

Előszó

A számítógépes grafika és animáció mára a számítástechnika egyik külön tudományává fejlődött. Valamennyi programozó, minden egyes szoftvergyártó cég arra törekszik, hogy terméke tetszetős és szép küllemű legyen. Jó példa erre a Microsoft cég *Windows* programja, amelynek kezelőfelülete felhasználóbarát, használata egyszerű. Nem kell tehát annak az okát keresni, hogy miért kell a grafika használatát és programozását ismerni, ez minden programozó számára egyértelmű.

Ez a könyv az IBM PC egyik legelterjedtebb grafikus kártyájának, a VGA (Video Graphics Adapter) kártyának a lehetőségeit mutatja be. A VGA és továbbfejlesztett változatai, az SVGA kártyák a PC-n egyre jobban terjednek, mára szinte minden más grafikus adaptert a háttérbe szorítottak. Sajnos a hardverelemek gyártói általában nem publikálják a különböző kártyák programozásához szükséges alapvető ismereteket, ezekhez csak a nagy szoftvergyártó cégek juthatnak hozzá. Az egyszerű programozó számára a fontos leírások elérhetetlenek. Talán ez az oka annak, hogy sok olyan szoftver lát napvilágot, amelynek grafikai képességeit a legnagyobb jóindulattal sem lehet elfogadhatónak nevezni.

A könyv programozókhoz szól, a benne szereplő programok megértéséhez elengedhetetlen a **Turbo Pascal** és az **assembly** nyelv igen alapos ismerete. Szükségünk lesz a két nyelv ötvözésére, mert Pascalban kényelmesen tudunk megoldani sok olyan feladatot, amely assembly szinten már nehézkes lenne, ugyanakkor az assembly nyelvet sem hagyhatjuk el, mert a VGA végső soron egy hardverelem, amelyet alacsony szinten tudunk igazán hatékonyan programozni. Szerencsére a Turbo Pascal 6.0-s és 7.0-s változatai támogatják a két nyelv összekapcsolását. A könyvben sorra vesszük a különböző VGA módokat, a beállításokat és a képszerkezeteket, szó lesz az egyszerű grafikus alapfeladatokról (pontkigyújtás, vonalrajzolás) és az összetett megjelenítő algoritmusokról, mint például a BOB-programozás. Megnézzük, hogyan lehet görgetni (scrollozni), és hogyan lehet a színekkel bűvészkedni.

Remélem, hogy a könyv példái és fejezetei hozzásegítik az Olvasót a VGA videoadapter megismeréséhez és az alapvetően regiszterorientált programozási filozófia elsajátításához.

László József

Mire lesz szükség?

A következőkre: egy IBM AT számítógépre, amelyben *legalább* 80286-os processzor tevékenykedik szüntelenül, és természetesen egy tetszőleges típusú VGA kártyára. Nem árt, ha van egy winchester a gépben, amelyre előzőleg a Turbo Pascal 6.0-t, a Turbo Pascal 7.0-t vagy a Borland Pascalt telepítettük már. Ha mindezekkel rendelkezünk, és a merevlemezen van elegendő szabad hely (1.2 M), akkor hozzá lehet fogni a lemezmelléklet installálásához. Az installálást egy egyszerű batch fájl (DOS parancsállomány) végzi. A merevlemezen lépünk be a megfelelő alkönyvtárba, ahol az anyagot tárolni és használni akarjuk, például: **c:\work**. Ezt követően írjuk be a következőt: **a:install**. A program betöltődik, majd szépen elkezdi átmásolni a programokat a lemezmellékletre a merevlemezre. A másolás eredményeképpen a következő könyvtárstruktúra keletkezik:

```
.....valami... \vgabook\  
                           charmode\  
                           g16mode\  
                           g256mode\  
                           palette\  
                           nonstd\  
                           trident\
```

A különböző könyvtárak a könyv különböző fejezeteihez tartoznak. Minden fejezet elején megtaláljuk annak a könyvtárnak a nevét, amelyikben a fejezet példaprogramjai találhatóak. A karakteres képről szóló fejezetben például így:



\CHARMODE

A példaprogramok mellett pedig egy



ikon mutatja a program nevét:



PELDA1.PAS

A lemezmelléklet átmásolása után még le kell fordítani a következő programokat:

```
charmode\crtplus.pas  
g16mode\grplus.pas  
g256mode\gr256.pas  
g256mode\gr256_2.pas  
trident\t8900c.pas
```


Ezek a fájlok a később használt *unitok* forrásai, ezért fordítás után érdemes mindenkinek a saját UNITS könyvtárába bemásolni a fordításkor keletkező .TPU fájlokat, hogy a továbbiakban a fordító el tudja érni azokat. A könyv fejezeteiben egyébként sorra részletesen is megismerjük majd ezeket a programokat. Ha mindezzel elkészültünk, akkor a lemez melléklettel a továbbiakban nem kell már foglalkozni, kezdődhet a munka. (Egy jó tanács: aki 6.0-s vagy 7.0-s Turbo Pascalt használ, az állítsa a fordítás céleszközét lemezre — *Compile/Destination Disk* —, néhány program ugyanis tetemes méretű memóriákat használ, és a szerkesztővel együtt nem mindig férnek el az alsó 640K memóriában.)

Miről is lesz szó a továbbiakban ? Az első négy fejezetben általános ismereteket szerzünk a VGA-ról, a regiszterkészletéről, a VGA BIOS szolgáltatásairól és a különböző képmódokról, képszerkezetekről. Ezek után terjedelmesebb példák segítségével megismerkedhetünk a háromféle videomód (karakteres, 16 és 256 színű videomód) programozásával. Minden egyes elméleti megközelítés után egy példa következik az adott feladat megoldására. A könyv nagy részében különböző példaprogramokat fogunk átvinni, mert gyakran jobb egy megoldást a gyakorlati megvalósítás alapján bemutatni, mint az amúgy is homályos feladatot elméleti megközelítésekkel tovább ködösíteni. A megoldás megértése így talán több időt vesz igénybe, de ha a gyakorlati megvalósítást megértjük, akkor az elmélettel is teljesen tisztában leszünk. Ezért az Olvasó ne ijedjen meg az első látásra zavarosnak tűnő, oldalnyi assembly listától, az is csak egy egyszerű program, amelynek megértése egyáltalán nem lehetetlen!

Ezek után szükség lesz még egy nagy lélegzetvételre, és tökéletesen szilárd, rendíthetetlen türelemre...

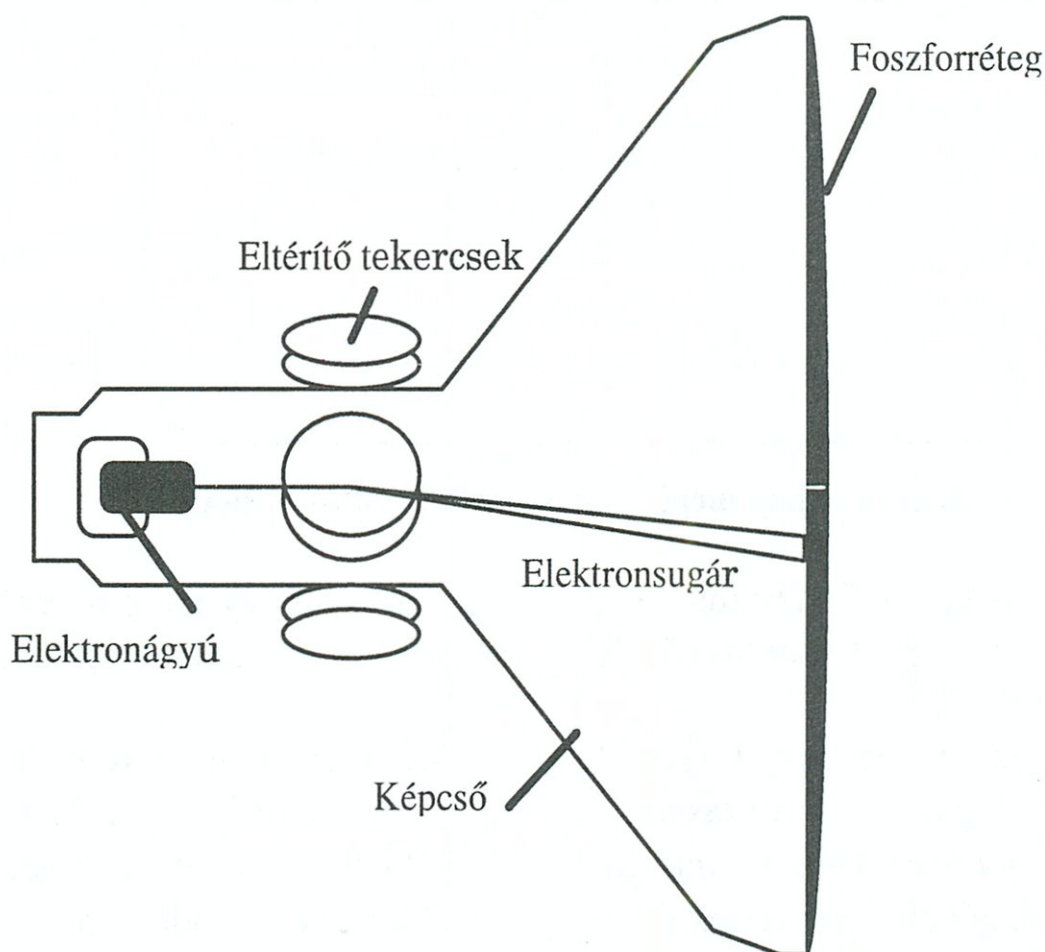
1. Alapfogalmak

A könyv további részeinek megértéséhez tisztáznunk kell néhány alapvető fogalmat és kifejezést, amelyek a televíziózással és a képmegjelenítéssel kapcsolatosak. Akik ebben a témában még nem jártasak, mindenképpen olvassák el ezt a fejezetet, mert a későbbiekben esetleg nem ismert fogalmakkal találkozhatnak. Nem törekszünk teljes műszaki leírásokra, csupán nagy vonalakban, néhány elvet ismertetünk.

1.1. A monitorok elvi működése

1.1.1. Monokrom képábrázolási módszer

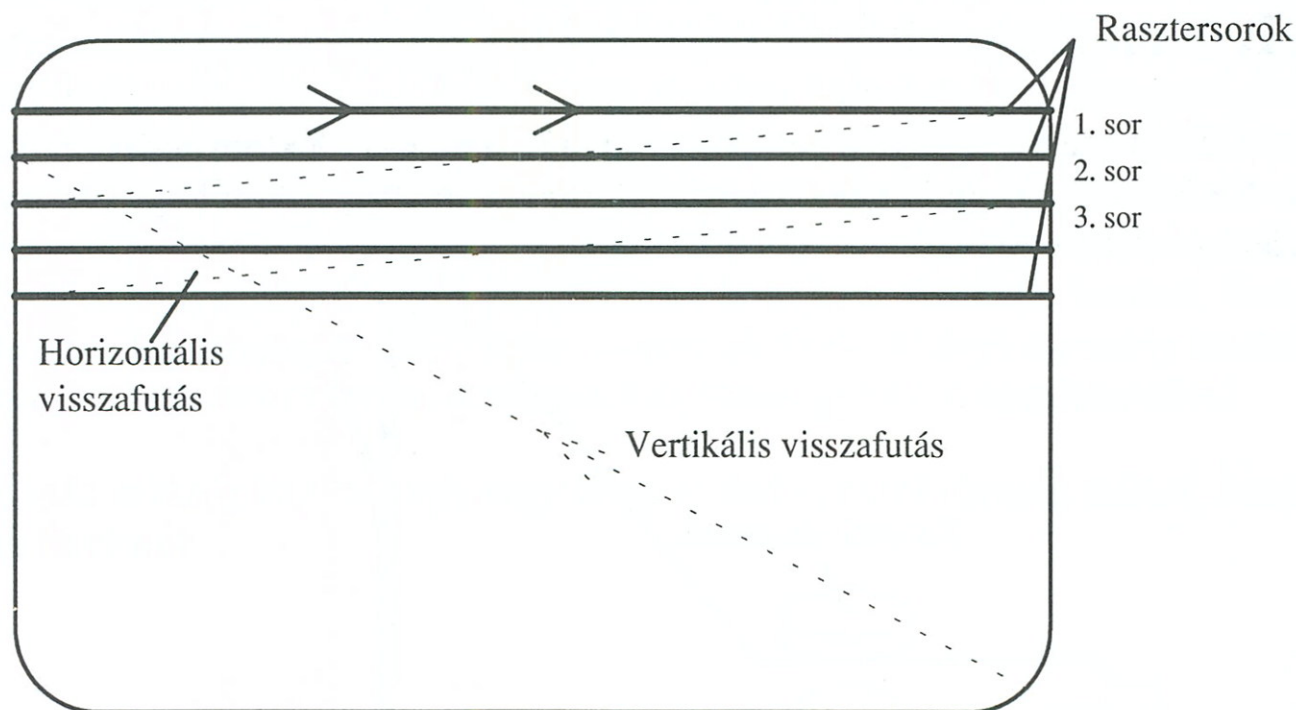
Egy szokványos tévékészülék képernyője nem más, mint egy elektronsugárcső. A kép felénk forduló oldala egy fényérzékeny foszforréteg, amelyet belülről egy elektronágyú elektronokkal bombáz.



1. ábra Az elektronsugárcső felépítése

Az elektronágyú folyamatos elektronsugarat bocsát ki, amely a képernyőn végigfutva érintkezik a foszforréteggel, és ezt az ütközést (az elektronok becsapódását a foszforba) mint fényjelenséget érzékeljük. A képcsőben van két eltérítő tekercs, amelyekkel a sugár megfelelően pozicionálni lehet a képen. Az eltérítés elve igen egyszerű: a sugár a két eltérítő tekercs terén vezetik keresztül, amely tekercsekre feszültséget kapcsolva az elektronsugár eltér az elektromágneses térerő hatására.

A kép rajzolása a bal felső pontból indul. A sugár elindul vízszintesen, és felrajzol egy vízszintes sort. A sor végén a sugárat kioltják, az visszafut ismét a kép bal szélére, de közben függőleges irányban lefelé mozdul. Ekkor megkezdődik a következő vízszintes sor kirajzolása. A tévékép úgynevezett félképváltásos módon rajzolódik ki, azaz az elektronsugár először a páratlan sorokat rajzolja fel, majd visszafut a kép elejére, és a páros sorok következnek.



2. ábra A kép megjelenítése (félképváltásos módszer)

Egy normál tévékép 625 vízszintes sorból épül fel, azaz egy ilyen tévékészüléken a legnagyobb függőleges felbontás 625 lehet, de a gyakorlatban ez ennél kisebb.

Ahhoz, hogy az emberi szem folyamatos mozgásnak érzékelje a képváltásokat, másodpercenként legalább 25 váltásnak kell lennie. A mozifilmek is 25 képkockából raknak össze egy másodpercnyi mozgást. Ha ennél kevesebbszer rajzolódik fel a kép, akkor már villogónak, remegőnek érzékeljük azt. Az egy másodpercre jutó képváltásokat az úgynevezett képváltási frekvenciával szokták megadni, ami Európában 50 Hz (50 félkép, azaz 25 teljes kép másodpercenként). A számítógépes monitorok ese-

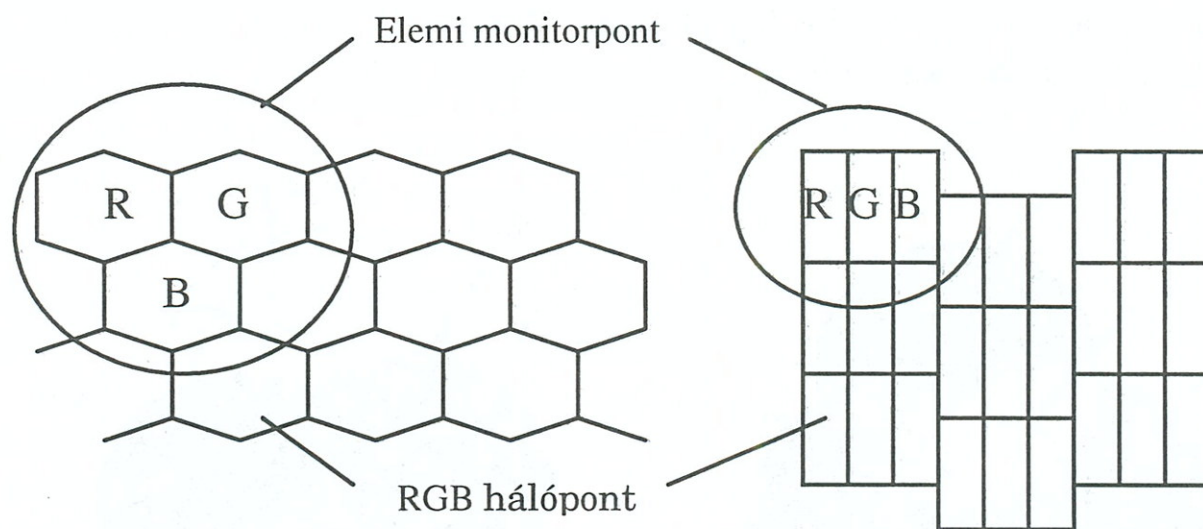
tében ez az érték általában nagyobb, már a CGA (Color Graphics Adapter) monitorok is 60 Hz-esek voltak, a normál VGA pedig már 70 Hz-en működik.

Mindebből a következő fogalmakat kell ismernünk:

- egy vízszintes sor, amelyet a sugár felrajzol a **rasztersor** (scan line);
- az a folyamat, amíg a sugár a kép jobb széléről visszatér a bal szélre a vízszintes vagy **horizontális visszafutás** (horizontal retrace);
- az a folyamat, amíg a sugár az utolsó sorból ismét a legelső elejére kerül a függőleges vagy **vertikális visszafutás** (vertical retrace);
- a **képváltási frekvencia** megadja az egy másodperc alatt felrajzolt félképek számát;
- **üres intervallumnak** nevezzük a képernyőnek azt a részét, ahol az elektronsugár végighalad ugyan, de már nem hordoz megjelenítési információkat. Ez a kép szélét jelenti, ahol már nincs semmiféle rajzolás. Megkülönböztetjük a függőleges üres intervallumot (ez a **vertical blanking**) és a vízszintes üres intervallumot (ez a **horizontal blanking**).

1.1.2. Színes képábrázolási módszer

A fent leírt képrajzolási folyamat monokrom monitorok illetve fekete-fehér tévékészülékek esetében használatos, a színes kép előállítása valamivel bonyolultabb. A foszforréteg három különböző típusból épül fel, és egy színháló található a képcsővön. A 3. ábra a két legelterjedtebb hálótípust mutatja be.



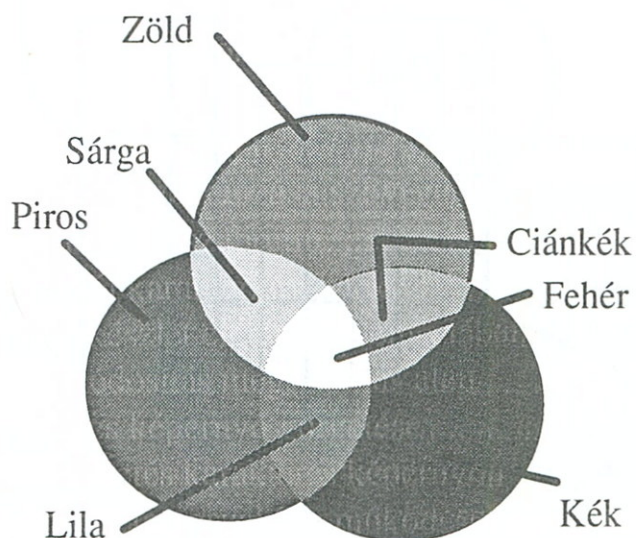
3. ábra A képcső elemi hálópontjai

A képcső nem egy, hanem három elektronágyút tartalmaz, három elektronsugár rajzolja a képet. A sugarak együtt haladnak, de mindegyik csak a saját elemi hálópontját világítja meg: a vörös sugár a vörös foszfort, a zöld a zöldet, a kék pedig a kéket. Így tulajdonképpen a háló összetartozó vörös—zöld—kék pontjai alkotják a képernyő egy *elemi monitorpontját*. Itt kell megemlítenünk a vízszintes felbontás fogalmát.

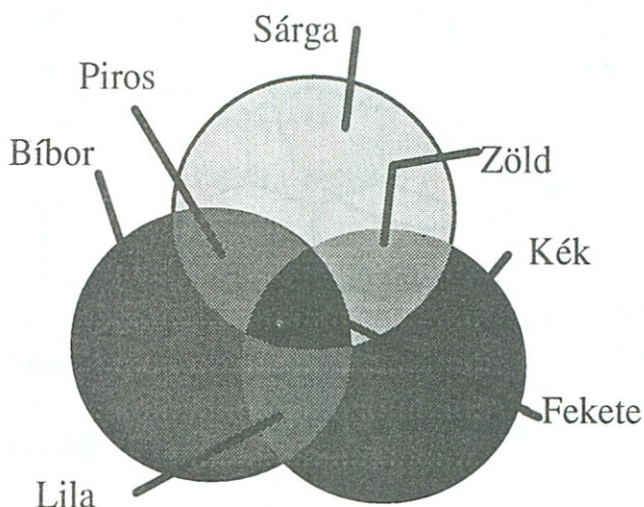
Látható, hogy a monitor legfeljebb csak annyi pontot tud megjeleníteni egymás mellett, amennyi a háló által képzett *elemi monitorpontok* száma vízszintesen. Ezzel eljuttunk a **pixel** fogalmához, amely nem más, mint a legkisebb képpont, amit a gép meg tud jeleníteni. A pixel nem feltétlenül egy elemi pont, függ a felbontástól az, hogy egy pixel hány elemi pontot tartalmaz, de a legkisebb megjeleníthető pixel legalább egy elemi pont. Másképpen: minden elemi pont pixel, de nem minden pixel elemi pont. A pixelek építik fel a számítógép képét. A pixelek formája felbontásfüggő, de a pixelek általában téglalap alakúak. Bár az ideális a négyzet alak lenne, a különböző felbontások miatt ez csak ritkán teljesül, nevezetesen akkor, ha a vízszintes és a függőleges felbontás aránya megegyezik a képernyő vízszintes és függőleges hosszának arányával.

1.2. Az RGB színkeverési módszer

A színes képcsövekben a színkeverés additív elven történik. Az additív színkeverés lényege, hogy a különböző színes fények egymással összeadva újabb színeket adnak. Ez a módszer nem tévesztendő össze a szubsztraktív elvű színkeveréssel, amelynek tipikus példája a különböző festékek keverése. Ha például egy kék és egy sárga festéket összekeverünk, akkor azok különbsége, vagyis zöld lesz az eredmény. Ha valaki az additív színkeverést akarja kipróbálni, akkor annak azt javasoljuk, hogy vegyen három színes izzólámpát (pirosat, zöldet és kék), és világítson velük egy adott helyre. A keverés a 4.a ábra szerint fog alakulni. (Fontos az azonos fényerősség.) A lemez mellékleten a **szinek.gif** fájl a 4. ábrát tartalmazza színesben.



4.a Additív színekörök



4.b Szubsztraktív színekörök

4. ábra A kétféle színkeverés

A három alapszín: vörös (**R**ed), zöld (**G**reen) és kék (**B**lue). Additív színkeverés esetén ebből a három alapszínből minden szín kikeverhető a feketétől a fehérig.

Nézzük meg, hányféle szín keverhető ki, ha mindhárom alapszínt csak kétállapotúnak (vagyis digitálisnak) feltételezzük:

R (vörös)	G (zöld)	B (kék)	Eredő szín
nincs	nincs	nincs	fekete
nincs	nincs	van	kék
nincs	van	nincs	zöld
nincs	van	van	ciánkék
van	nincs	nincs	piros
van	nincs	van	lila
van	van	nincs	sárga
van	van	van	fehér

A CGA monitorok esetében ezt a módszert alkalmazták, kiegészítve egy intenzitásállapottal, amely szintén kétállapotú volt, így a megjelenő színek számát 16-ra növelték (gyakorlatilag csak 15-re, mert a fekete nagy intenzitása is csak feketét adott). Ez volt az úgynevezett RGBI monitorok színkeverési rendszere, ahol minden jel digitális formában jutott el a géptől a monitorig.

A további fejlesztés eredményeképpen létrejött EGA (Enhanced Graphics Adapter) monitoroknál minden színtelephoz 2 bitet rendeltek, így az alapszínek négyállapotúak lettek. Ebben az esetben a kikeverhető színek száma: $4 \cdot 4 \cdot 4 = 64$ volt. Általánosan elmondható, hogy egy RGB keverési módszerrel az előállítható színek száma az egyik alapszín lehetséges állapotainak köbe. Ma az úgynevezett **True Color** (valóságos színeket megjelenítő) monitorkártyákon mindhárom alapszín 8-bites, azaz 256 árnyalatú lehet, így az előállítható színek száma **16 777 216** (például az Amiga 1200-as gép AGA áramköre az RGB komponenseket 8 biten ábrázolja).

A VGA monitorok esetében a színtelephoz analóg feszültségértékek. A VGA kártya 6-bites RGB regisztereket tartalmaz ($64 \cdot 64 \cdot 64 = 262\,144$ szín), az ezekben található értékek egy D/A átalakítóra kerülnek, onnan pedig a monitor analóg bemenetére. A színkeverésre nézzünk egy rövid példaprogramot, amely az RGB regiszterek programozását is bemutatja:



PELDA1.PAS

Program Szinkeveres;

Uses crt;

var i:integer;

Procedure SetRGB(Pal,Red,Green,Blue:byte);

Begin { Egy RGB regiszter beállítása }

Asm

```

    mov ax,1010h
    mov bl,Pal    { A palettaregiszter száma a BL-be }
    xor bh,bh
    mov dh,Red    { Piros komponens a DH-ba }
    mov ch,Green  { Zöld komponens a CH-ba }
    mov cl,Blue   { Kék komponens a CL-be }
    int 10h       { A BIOS RGB-beállító funkciójának hívása }

```

End;

Delay(10);

End;

BEGIN

Clrscr;

for i:=0 **to** 63 **do** SetRGB(0,i,0,0); { Fekete-piros átmenet }

Delay(500);

for i:=0 **to** 63 **do** SetRGB(0,63,i,0); { Piros-sárga átmenet }

Delay(500);

for i:=0 **to** 63 **do** SetRGB(0,63,63,i); { Sárga-fehér átmenet }

Delay(500);

for i:=63 **downto** 0 **do** SetRGB(0,63,i,63); { Fehér-lila átmenet }

Delay(500);

for i:=63 **downto** 0 **do** SetRGB(0,i,0,63); { Lila-kék átmenet }

Delay(500);

for i:=63 **downto** 0 **do** SetRGB(0,0,0,i); { Kék-fekete átmenet }

Writeln('Kész!');

Readln;

END.

A SetRGB eljárás a VGA kártya egyik RGB regiszterét állítja be a VGA BIOS beállító rutinját felhasználva. Az első ciklus feketéből pirosba viszi át a háttérszínt. Ezt követően a pirosához hozzáadjuk a zöld komponenszt, ekkor kapjuk a sárga színt. Ehhez hozzáadva a kék komponenszt fehér lesz az eredmény.

Most a fehérből elkezdjük levonni a színeket. Először a zöldet (ekkor lila az eredmény), majd a pirosat (most marad a kék). Végül a kék komponenst is levonjuk, így eljutunk az eredeti fekete állapothoz. Figyeljük meg, hogy az egyes színkomponenseket csak a 0..63 intervallumban változtatjuk (mivel a színek 6-bitesek), az ennél nagyobb értéknek nincs értelme, a kártya a bájtok felső két bitjét figyelmen kívül hagyja. Az RGB regiszterek programozásáról bővebben **A paletta programozása** című fejezetben lesz szó.

2. Képernyőszerkezetek

A számítógép képernyője a memóriában helyezkedik el, mint különleges felhasználási memóriaterület. A videokártya valamilyen módszerrel (általában DMA hozzáféréssel) rendszeresen kiolvassa ezt a területet, és az ott található adatokat valamilyen formában megjeleníti. A különböző képernyőmódokhoz különböző megjelenítések tartoznak, ezek határozzák meg, hogy egy adott esetben milyen a kiolvasás logikája, azaz milyen a képernyő szerkezete. A VGA esetében ez a különleges videoterület az A000:0000...B000:FFFF címen lévő memóriaterület. Az itt található adatok jelennek meg a képernyőn, de általában nem a teljes 128K-s terület, hanem annak valamelyik kisebb része.

2.1. A karakteres kép felépítése

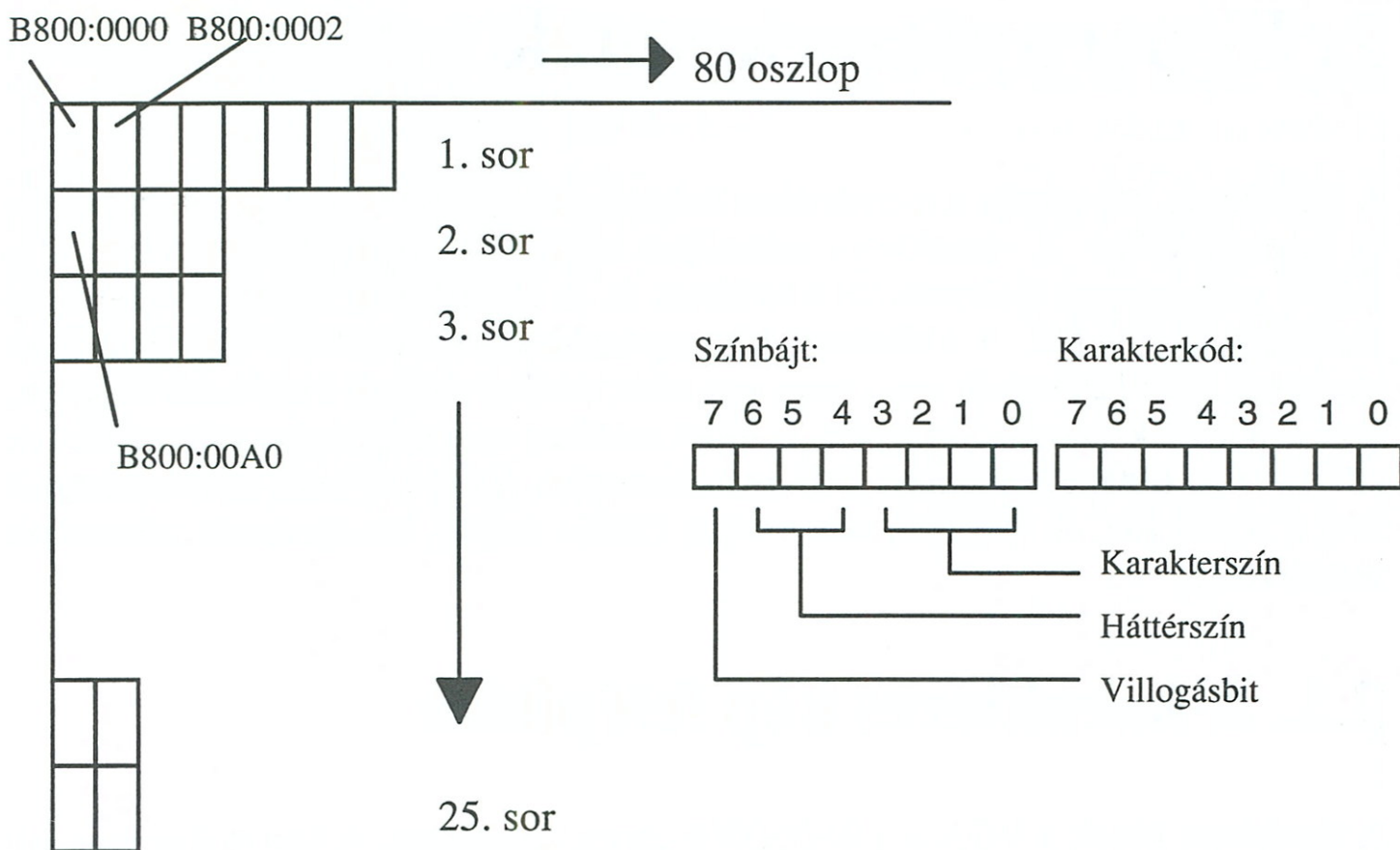
A karakteres képek a VGA-n a B800:0000 címen kezdődnek. A kártya ilyenkor szó címezésmódban (word addressing mode) van, egy karaktert egy 16-bites szó határoz meg. Az alsó bájt a megjelenő karakter kódját, míg a felső bájt a karakter színét (attribútum állapotát) határozza meg. Ha tehát a B800:0000 címre beírunk egy 16-bites kódot, akkor a karakteres kép bal felső sarkában megjelenik a kódhoz tartozó karakter a megadott színnel. Alapesetben a képernyőn 80x25 karakter helyezkedik el, tehát egy ilyen kép 2000 word memóriát, azaz 4000 bájtot igényel. Két karakter vízszintes címtávolsága tehát 2 bájt, függőlegesen pedig 160 bájt. Egy adott karakter pozíciójából a hozzá tartozó képcímet ezek alapján a következőképpen határozhatjuk meg:

$$\text{Képcím} := X\text{koordináta} * 2 + Y\text{koordináta} * 160$$

Mivel a képen nemcsak 80, hanem 40 karakter is lehet, ezt a képletet általánosítanunk kell az alábbi formára:

$$\text{Képcím} := X\text{koordináta} * 2 + Y\text{koordináta} * \text{egy sor hossza}$$

Az *egy sor hossza* paraméter 40 vagy 80 az adott karakteres módtól függően. Az 5. ábra alapján mindez érthetőbbé válik.



5. ábra A karakteres kép szerkezete

Nézzünk egy rövid példaprogramot! A program mind a 256 karaktert kiírja a képernyőre közvetlen memóriába írással:

Program Karakterek;

Var i:byte;

BEGIN

For i:=0 **to** 255 **do** mem[\$b800:i*2] :=i; { Minden karakter kiírása a képre }

Readln;

END.

A memóriába írást a **mem[]** tömbbel oldjuk meg. Figyeljük meg, hogy a program csak minden második bájtot írja át, azaz csak a karakterkódot módosítja, a színtérülethez nem nyúl. Módosítsuk úgy a programot, hogy a karaktereket egy adott színnel, pl. sárgával írja ki. Az 5. ábrán látható a színbájt (attribútum bájt) felépítése. Az alsó 4 bit adja meg az írás színét, a következő 3 bit a háttér színét, míg a 7. bit a villogást kapcsolja ki-be. A színbájt ettől eltérően is működhet (nevezetesen a 7. bit a villogás helyett a háttérszín összetevője is lehet, ekkor 16 színű írás és 16 színű háttér használható), de az alapbeállítás mindig a „villogás és 8 háttérszín” eset. A színekre nézve

is van egy alapbeállítás, amit természetesen felül lehet írni az RGB regiszterek közvetlen átprogramozásával.

Az alapbeállítású színek a következők:

0	0000	fekete
1	0001	kék
2	0010	zöld
3	0011	ciánkék
4	0100	piros
5	0101	lila
6	0110	barna
7	0111	világosszürke
8	1000	sötétszürke
9	1001	világoskék
10	1010	világoszöld
11	1011	világoscián
12	1100	világospiros
13	1101	világoslila
14	1110	sárga
15	1111	fehér

Ha tehát minden karaktert sárgán akarunk megjeleníteni, akkor a színbájtokat 14-re, azaz \$0E-re kell állítani. Mivel a színbajt a szó felső bájtja, ezért ezt meg kell szoroznunk 256-tal, amit legegyszerűbben az utána írt nullákkal tehetünk meg: $\$0E * 256 = \$0E00$

Program Karakterek;

Var i:byte;

BEGIN

For i:=0 **to** 255 **do** memw[\$b800:i*2] := \$0e00+i; { Most sárga lesz }
Readln;

END.

Most módosítsuk úgy a programot, hogy egy adott karakter jelenjen meg minden lehetséges színkombinációban! Legyen a megjelenítendő karakter a @ jel, ennek a kódja 64, azaz \$40. A programban módosítsuk a memóriába írást a következőre:

memw[\$b800:i*2] := 256*i+\$40;

Ezek után meg kell jelennie a 256 darab @ karakternek az összes lehetséges színnel.

A karakteres képernyő programozásáról egy teljes külön fejezet szól majd, de hogy az assembly programozásra is legyen példa, nézzünk egy egyszerű eljárást, amellyel adott pozícióba adott színű karaktert tudunk írni. Az eljárás helyettesíti a `gotoxy(x,y);writeln('x');` utasításpárt.

Procedure PrintXY(x,y:word;c:char;z:byte);**Assembler;**

Asm

```

push    ES                {Tároljuk az ES-t                }
mov     ax,0b800h
mov     ES,ax             {Az ES-t a képernyőre állítjuk    }
mov     ax,160            {40 karakteres mód esetén ez 80 lesz }
mul     y                {Címszámítás: egy sor 160 bájtól áll }
mov     di,x
shl     di,1              {Cím = x*2 + y*160                }
add     di,ax             {Az ES:DI most a karakteres képre mutat }
mov     al,c              {Az alsó bájt lesz a karakter kódja   }
mov     ah,z              {A felső bájt lesz a karakter színe   }
stosw
pop     ES                {Visszahozzuk az ES eredeti értékét }

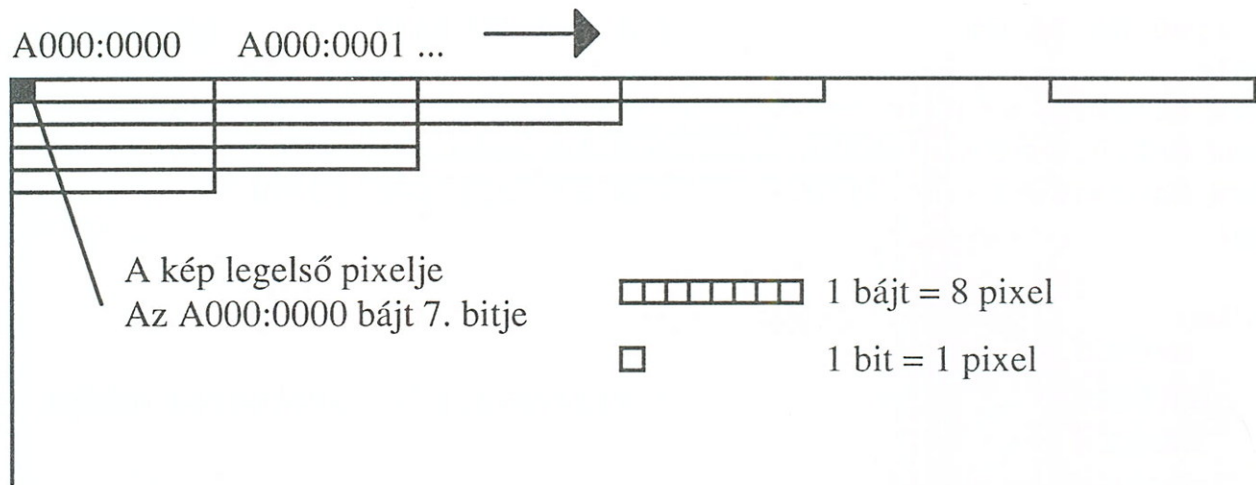
```

End;

Az eljárás a `gotoxy` eljárástól eltérően a 0 pozícióval kezdi a koordinátákat, ezért a lehetséges intervallum egy 80×25-ös kép esetén $x=0..79$, $y=0..24$. Az eljárás nem ellenőrzi a koordinátaértékeket, ezért a híváskor vigyázni kell, nehogy illegális értéket adjunk át.

2.2. A monokrom grafikus kép szerkezete

A grafikus képernyők bittérképes képek. Ez azt jelenti, hogy a pontok (pixelek) leírására a memória egy vagy több bitjét használjuk fel. A grafikus képet pixelek építik fel, ezek egy monokrom kép esetén kétállapotúak lehetnek: kigyújtottak (tinta színűek) vagy kioltottak (háttér színűek). A VGA-n a grafikus képek az A000:0000 címen kezdődnek, és egy bájt 8 pixelt tartalmaz, azaz minden pixelhez egy bit van hozzárendelve. Ha a megfelelő bájt megfelelő bitjét 1-be billentjük, akkor a bithez rendelt pixel kigyújtott állapotba kerül.



6. ábra A monokróm grafikus kép szerkezete

A bájtok természetesen címfolytatosan követik egymást. Az első pixel a képen az A000:0000 bájt 7. bitje, a második pixel ugyanennek a bájt-nak a 6. bitje és így tovább. A felbontástól függ, hogy hány bájt épít fel egy vízszintes sort, pl. 640 pontos vízszintes felbontás esetén $640/8$, azaz 80 bájt alkot egy sort. A 81. bájt lesz a 2. sor legelső bájtja, azaz a sorok is címfolytatosan követik egymást. Más videokártyák esetén előfordulhat, hogy a sorok nem címfolytatosak (például a Hercules kártya esetében). A sorok száma és az egy sorban elhelyezkedő bájtok száma megadja, hogy a képernyő hány bájt hosszú a memóriában. Tekintsük példaképpen a 0Fh sorszámú videomódot (lásd később a BIOS leírásánál), amely 640×350 -es felbontásban dolgozik monokrom módban. Ez a felbontás 80 bájtot igényel egy sorban, és 350 sorból építi fel a képet. Tehát a szükséges memória: $80 \times 350 = 28\,000$ bájt. Érdeemes megfigyelni, hogy a szabványos képfelbontások minden esetben egész számú bájtot tartalmaznak vízszintes irányban:

320×200	40 bájt/sor
640×200	80 bájt/sor
800×600 (SVGA)	100 bájt/sor

A következő program a mem[] tömb felhasználásával ír a képre:



PELDA2.PAS

Program Mono_grafikus_kep;

Var i:byte;

BEGIN

Asm

mov ax,000fh

int 10h

{ Átváltás a 640x350/mono módra

End;

```

for i:=0 to 79 do                { 1 sor 80 bájt }
  Begin
    mem[ $a000:i] :=$ff;          { Az 1. sor folytonos vonal }
    mem[ $a000:80*9+i] :=$33;    { A 10. sor szaggatott vonal }
    mem[ $a000:80*19+i] :=$7d;   { A 20. sor pontvonal }
  End;

Readln;

Asm
  mov  ax,0003h                  { Visszaváltás karakteres módra }
  int  10h
End;

END.

```

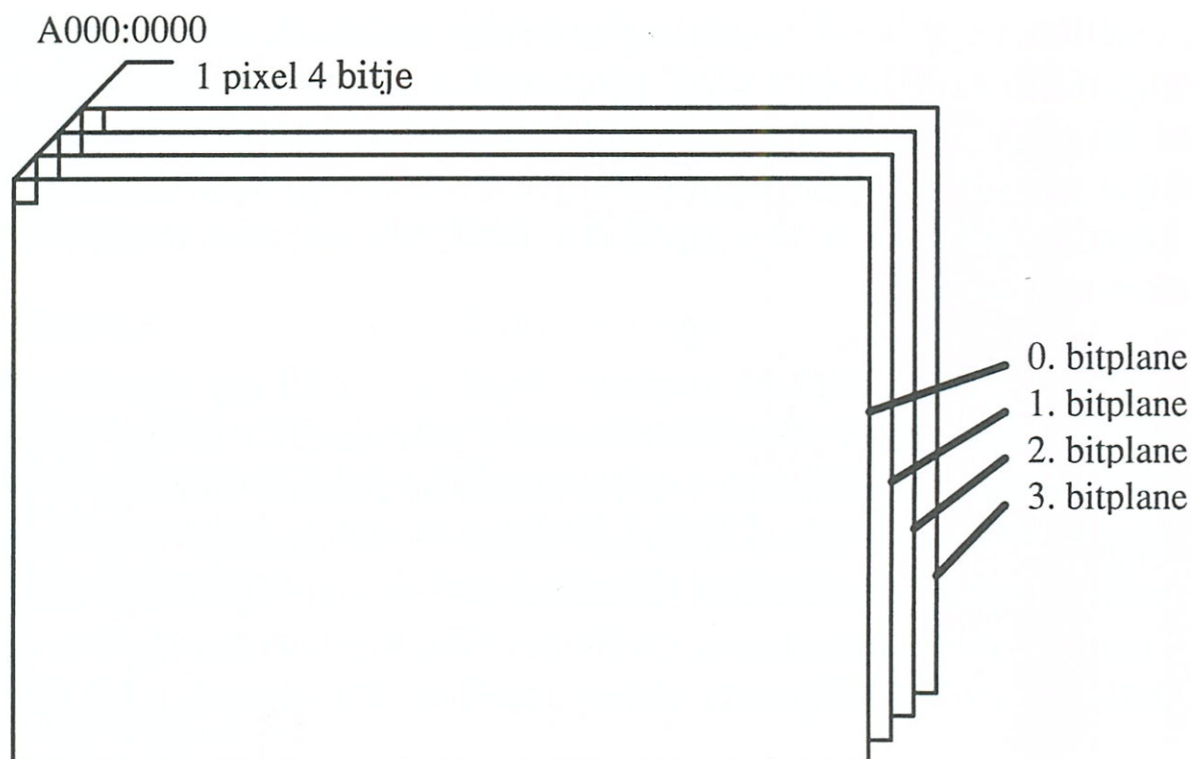
A program három vonalat rajzol a képre. Az 1. sort olyan bájtokkal tölti fel, amelyeknek minden bitje 1-es, azaz 255-tel. A második vonalat \$33-mal tölti fel, amelynek a bináris képe: 00110011, ez egy szaggatott vonalat eredményez. Végül a harmadik vonal bitképe: 01111101, ez lesz a pontvonal. A program elején és végén lévő **int 10h** megszakítással a megfelelő videomódot állítjuk be, erről bővebben a BIOS leírásánál lesz szó.

2.3. A 16 színű grafikus kép szerkezete

A monokrom kép leírásakor láthattuk: egy bit egy pixel állapotát határozza meg, oly módon, hogy megadja, vajon az adott pont tinta vagy háttér színű legyen-e. Ha a kép színes, akkor már nem elegendő egyetlen bit egy pixelhez. Tizenhat szín esetén 4 bit kell egy pixel leírásához. A képernyő memóriában való leképezésére két lehetőség kínálkozik:

1. Mivel egy bájt 8 bitet tartalmaz, ez 2 pixel leírásához elegendő. Legyen tehát minden 2 pixel egy bájtban, és ezekből épüljön fel a képernyő egésze.
2. Ha egyetlen bittérkép 2 állapotot tud tárolni, akkor legyen 4 egymást követő bittérkép, és az ezekben található bájtok összessége ábrázolja a teljes képet.

Mindkét megoldásra találunk példát a különböző számítógépeknél; a VGA esetében a második formátumot alkalmazzák. A színes képet úgynevezett **bitplane**-ek alkotják, amelyeket úgy kell elképzelni, mintha egymás mögött lenne négy képernyő.



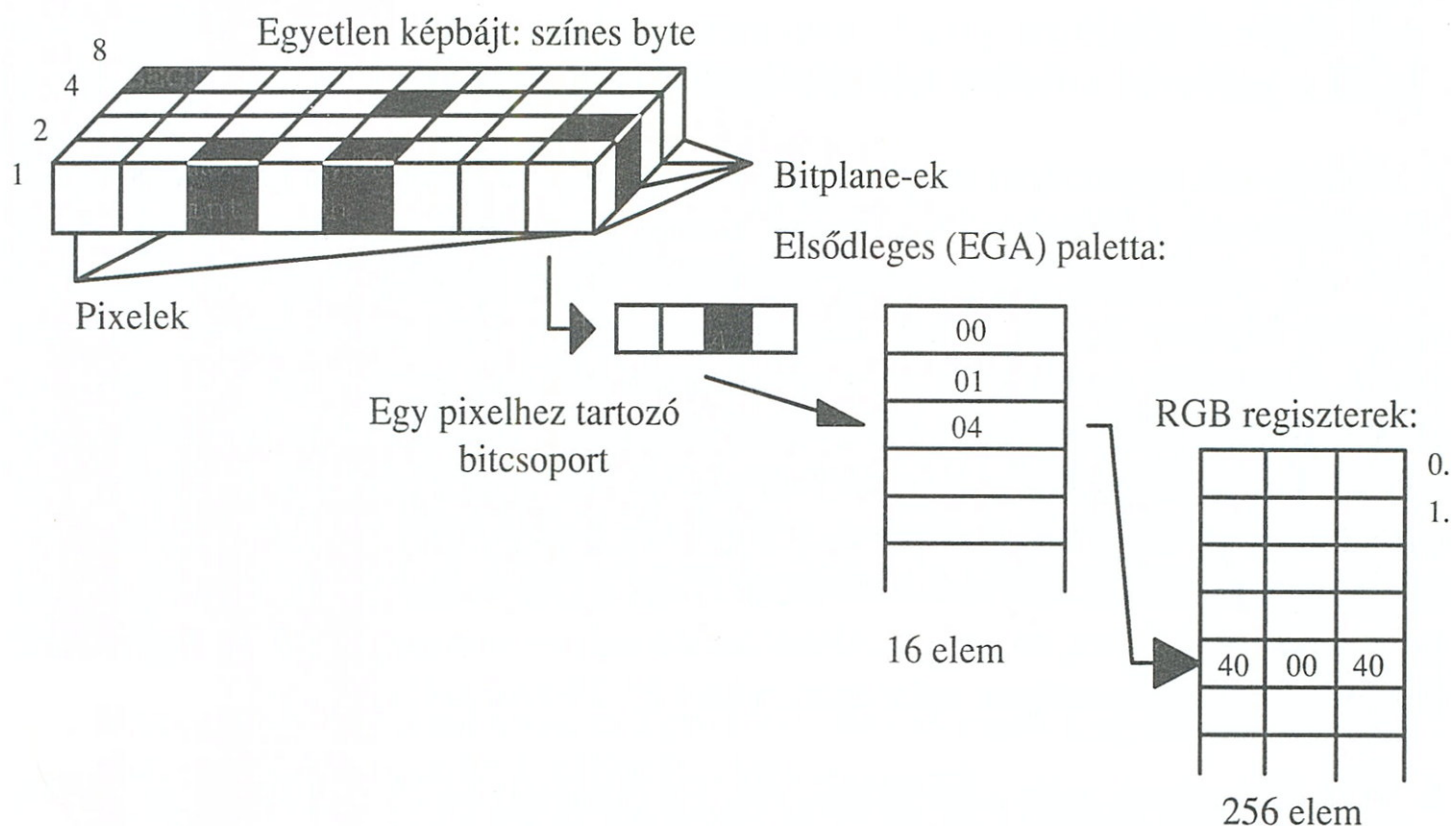
7. ábra A 16 színű grafikus kép szerkezete

A bitplane-nek ugyanolyan tulajdonságai vannak, mint a monokrom képnek: a bájtok címfoltyonosak, a különböző bitplane-ek azonos címen levő bájtjainak azonos számú bitjei egy adott pixelhez tartoznak. Az alapvető különbség az, hogy nem elegendő egyetlen bitplane átírása a kép megváltoztatásához. Más számítógépek esetében (Atari, Amiga) az összes bitplane a memóriában található egymás mögött. Ennek az ábrázolásnak az az előnye, hogy közvetlen memóriába írással a teljes kép megváltoztatható, hátránya viszont a nagy memóriaigény. Az előbbi példából kiindulva: egy 640×350 -es 16 színű kép $80 \times 350 \times 4 = 112000$ bájtot igényel. A fejezet elején említettük, hogy a képterület kiolvasása általában DMA ciklusok alatt zajlik. Ez a kiolvasási mód persze elég gyors, de nagy területek esetén már ez sem megfelelő, ha figyelembe vesszük, hogy a képterületet másodpercenként 25-35-ször kell kiolvasni.

A VGA kártyáknak saját memóriájuk van. Ez minimálisan 256 Kbájt, de az SVGA kártyákon nem ritka az 1 Mbájt memória sem. Hogy a processzor saját memóriájából a képernyő ne foglaljon sok helyet, és a DMA ciklusok se nyúljanak túl hosszúvá, ezért a VGA kártyákon egy sajátos megoldást alkalmaznak: valamennyi bitplane ugyanazon a memóriacímen érhető el (az A000:0000 címen), és lehetőség van a bitplane-ek együttes írására. A kártyát I/O portok segítségével kapcsolhatjuk a megfelelő írásmódba (ebből több is van). A 3. fejezet ezeknek az I/O portoknak (VGA regisztereknek) a leírását tartalmazza.

A színek előállítása egy belső paletta segítségével történik (8. ábra). Mivel a VGA kártya kompatibilis az EGA és a CGA kártyákkal, a paletta kialakítását úgy kellett megoldani, hogy az megfeleljen az EGA formátumnak is. Ezért az egy pixelhez tartozó színt dupla indexeléssel lehet meghatározni. A 8. ábrán egyetlen képernyőbájt látható. A képbájt valójában 4 bájt együttese, amelynek egymás mögötti bájtjai a bitplane-eken találhatók.

A kártya kiolvassa minden bitplane megfelelő bitjét, így előáll egy négybites mutató, amely megadja, hogy melyik EGA szín tartozik a pixelhez. Ha ez egy EGA kártya lenne, akkor a mutató már közvetlen 3*2 bites RGB információt mutatna, de a VGA-n ez csak egy újabb index a valódi RGB palettához. Az itt található RGB szín lesz a pixel színe. Ez a dupla indexelés nem túl szerencsés, de a kompatibilitás miatt szükséges volt ezt a módszert alkalmazni. (A Turbo Pascal *Graph* egységében található egy SetPalette és egy SetRGBpalette eljárás, amellyel beállíthatjuk a két különböző palettát.)



8. ábra A színérték meghatározása

Gyűjtsuk ki a képernyő négy sarkán elhelyezkedő pixeleket 4 különböző színnel egy 320×200-as 16 színű képernyőn (\$0D sorszámú videomód)! A négy sarokban levő pixelek memóriacímei és a bitpozíciók:

Bal felső sarok: az A000:0000 bájt 7.bitje

Jobb felső sarok: az A000:0027 bájt 0.bitje

Bal alsó sarok: az A000:1F18 bájt 7.bitje

Jobb alsó sarok: az A000:1F3F bájt 0.bitje

A színezést a VGA *bitplane* írásmaszk regiszterével oldjuk meg, amely megadja, hogy melyik bitplane-eket lehet írni egyszerre. A regiszter alsó 4 bitje határozza meg az írható bitplane-eket. A regiszter a \$03C4, \$03C5 portokon keresztül érhető el. A színértékek alapbeállítása megfelel a karakteres képszerkezetnél leírtaknak.



PELDA3.PAS

Program Sarkok_szinezese;

BEGIN

Asm

mov ax,000dh

int 10h { A 320x200/16 mód beállítása }

End;

Portw[\$3c4] := \$0f02; { A plane maszk: 1111 (0f) }

Mem[\$a000:\$0000] :=128; { A bal felső sarok fehér }

Portw[\$3c4] := \$0e02; { A plane maszk: 1110 (0e) }

Mem[\$a000:\$0027] :=1; { A jobb felső sarok sárga }

Portw[\$3c4] := \$0d02; { A plane maszk: 1101 (0d) }

Mem[\$a000:\$1f18] :=128; { A bal alsó sarok lila }

Portw[\$3c4] := \$0c02; { A plane maszk: 1100 (0c) }

Mem[\$a000:\$1f3f] :=1; { A jobb alsó sarok piros }

Readln;

Asm

mov ax,0003h

int 10h { A karakteres mód visszaállítása }

End;

END.

Az alábbi program 15 vízszintes színes csíkot húz. Ez a program is a bitplane írásmaszk regisztert használja. (Bővebben a regiszterek leírásánál lesz róla szó.)



PELDA4.PAS

Program Szines_Grafikus_Kep;

Var i,j:integer;

BEGIN

Asm

mov ax,000dh

int 10h { Átváltás a 320x200/16 módba }

End;

for j:=1 to 15 **do**

Begin

Port[\$3c4] :=\$02;

Port[\$3c5] :=j; { Bitplane írás engedélyezése }

for i:=0 to 39 **do** { Egy sor feltöltése 255-tel }

Mem[\$a000:j*40*3+i] :=255; { Egy sor 40 bájt }

End;

Readln;

Asm

mov ax,0003h { Visszatérés karakteres módba }

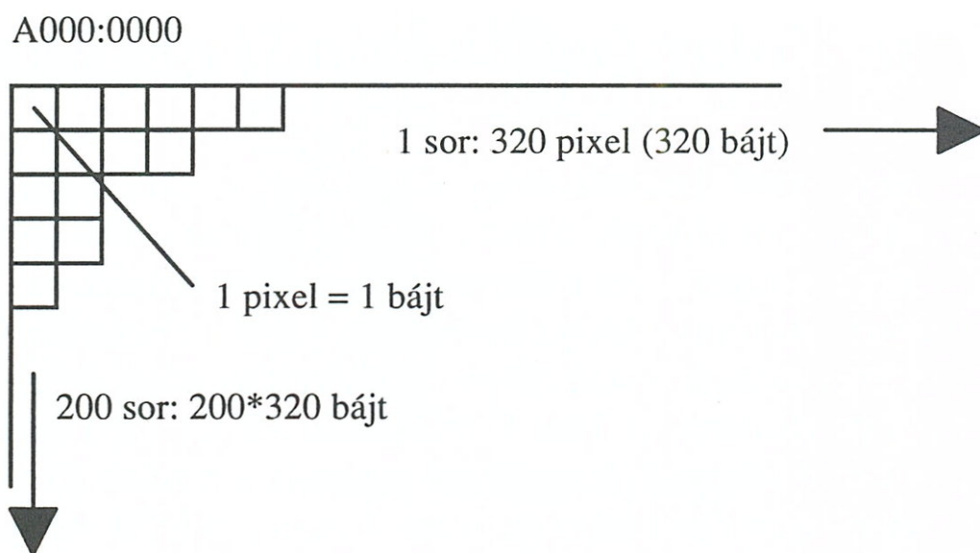
int 10h

End;

END.

2.4. A 256 színű grafikus kép szerkezete

A 256 színű kép a VGA kártya \$13-as számú üzemmódja, a felbontás 320×200 pixel. Minden egyes pixel 256 színű lehet, azaz ebben az esetben egy pixelhez 8 bitet kell hozzárendelni. Mivel egy pixelhez egy bájtot lehet rendelni, itt a bitplane technika már nem lenne ésszerű megoldás, így a képmemória egyszerűen bájtonként tárolja a képet. A kép memóriagénye $320 \times 200 = 64\,000$ bájt, a kezdőcím itt is az A000:0000. Az egyes bájtok mutatók az RGB regiszterkészletre, az EGA paletta nem játszik szerepet. Ez a képformátum egyszerű és gyorsan kezelhető, a pont- és a vonalrajzolás könnyen megoldható rajta. Ezt a képformátumot nevezzük *normál bájt/pixel* formátumnak.



9. ábra A 256 színű kép szerkezete

A következő program az alapbeállítású paletta minden elemét megjeleníti a képernyőn:



PELDA5.PAS

```
Program VGA256_mod;
```

```
  Var i,j:word;
```

```
BEGIN
```

```
  Asm
```

```
    mov ax,0013h
```

```
    int 10h
```

```
{ Átváltás a 320x200/256 módba }
```

```
End;
```

```
for j:=0 to 255 do
```

```
{ 256 db függőleges színes vonal }
```

```
  for i:=0 to 199 do
```

```
{ j->X i->Y pozíció, j-> szín }
```

```
    mem[ $a000:j+i*320] :=j;
```

```
Readln;
```

```
{ Várakozás az ENTER lenyomására }
```

```
  Asm
```

```
    mov ax,0003h
```

```
{ Visszatérés a karakteres módba }
```

```
    int 10h
```

```
End;
```

```
END.
```

2.5. A True Color (valós színű) grafikus kép

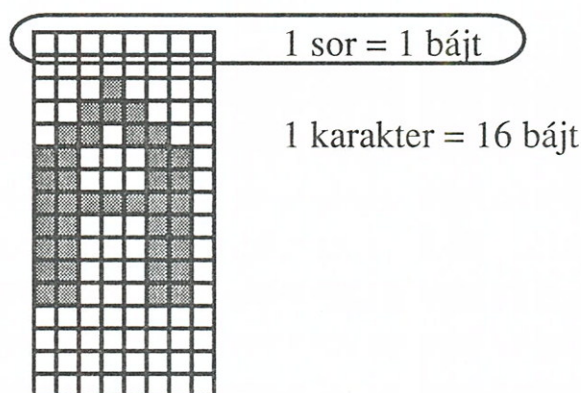
Az eddigi képszerkezeteknél láthattuk, hogy a szín kiválasztása mindig valamilyen előre definiált paletta szerint történik. Mind a 16 színű, mind a 256 színű kép esetében a pixelhez rendelt bitsorozat nem a közvetlen színadatot tartalmazza, hanem csak egy mutatót a palettára, ahonnan a kártya kiolvassa a valódi színösszetevőket. A True Color képszerkezet nem tartozik a szabványos VGA módok közé, de a teljesség kedvéért érdemes ezt is megismerni. A True Color (valódi színes kép) elnevezés azt jelenti, hogy a képet tároló adattömb közvetlenül megadja a pixelek RGB összetevőit. Így minden pixel tetszőleges színű lehet az összes lehetséges RGB variáció közül. (A 256 színű kép a VGA 262 144 színvariációjából csak 256-ot tud megjeleníteni egyszerre.) Ez persze azt jelenti, hogy a kép egy pixeléhez jóval több bitet kell hozzárendelni, mint más esetekben. Ha pl. a normál VGA kártya ismerné a True Color üzemmódot, akkor egy pixelhez 3×8 bit kellene. A mai fejlett True Color kártyák általában 8 biten tárolnak egy RGB komponenst, azaz a teljes RGB információ 24 bitet igényel. A True Color képek gyakori elnevezése, a „24-bites kép” is erre utal. Ezek a képek igen nagy memóriát igényelnek, mivel 1 pixel 3 bájtban tárolható el. Egy 1024×768 -as felbontású True Color kép memóriagigája **2 359 296** bájt, azaz 2.3 Mbájttal. Egy ilyen kép minősége már eléri egy színes fénykép minőségét. A True Color képek nagy hátránya, hogy az animáció és a rajzolás igen nehéz, egyrészt a kép szerkezete, másrészt a memória nagysága miatt. A True Color technikát ezért általában állóképek ábrázolására használják.

(Természetesen egy profi videostúdióban vannak olyan számítógépek, amelyek valós időben képesek kezelni a True Color képet.)

2.6. A karakter bittérképek

A karakteres kép szerkezetének leírásakor láthattuk, hogy egy bájt egy teljes karaktert tartalmaz. De a karakter is csak pixelekből épül fel, így annak formáját is tárolni kell valahogyan. A VGA többféle karakterkészletet tud használni a beállított karakteres módtól függően. Egy karakter felépítésére az úgynevezett *karakterboxot* használjuk, amelyben a karakter bittérképét tároljuk.

Az "A" betű bittérképe:



10. ábra Karakter bittérkép

A karakterek magassága videomódfüggő, de a tárolt szélesség minden esetben 8 bit. A VGA ROM (a VGA kártyán lévő csak olvasható memória) háromféle magasságú karaktereket tartalmaz: 8×8-as, 8×14-es és 8×16-os készletet. A megjelenített karakterbox mérete állítható, szélessége még egy pluszbittel megnövelhető, így 9 pixeles karaktereket tudunk használni (ez az alapbeállítás), de a 9. bitet nem tároljuk, az mindig háttér színű. A karakterboxok a videomemóriában találhatók az A000:0000 címen (emlékezzünk vissza, hogy a karakteres képek a B800:0000 címen kezdődnek), de nem lehet egyszerű memóriairással és -olvasással elérni azokat. A VGA BIOS tartalmaz olyan megszakításokat, amelyekkel a karakterkészlet beállítható és lekérdezhető. A karakteres mód programozását leíró fejezetben mindezekre fény fog derülni.

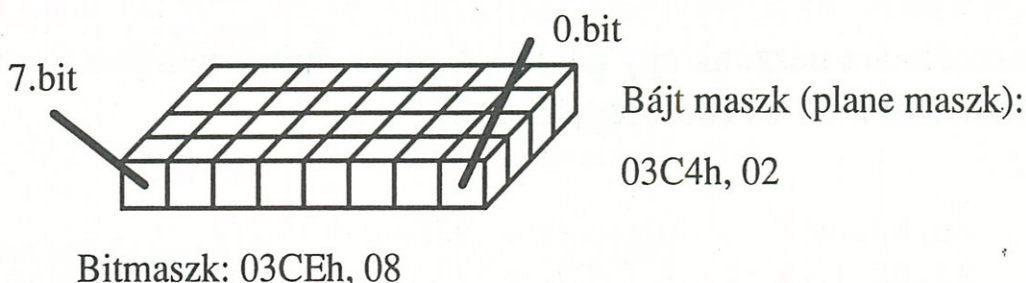
3. A VGA kártya regiszterkészlete

Ez a fejezet a szabványos IBM VGA kártya regisztereit mutatja be. A regiszterek több kisebb csoportra oszthatók, mint például szín- és palettaregiszterek, elektronsugár-vezérlő regiszterek, módbeállító és státuslekérdező regiszterek. A regiszterek elérhetők a Pascal **PORT** és **PORTW** különleges tömbjein keresztül vagy Assembler programok segítségével. A programozáskor nem szabad megfeledkezni két alapvető követelményről:

1. Ne írjunk a regiszterekbe illegális értékeket, mert ez a hardver meghibásodásához vezethet. Ez persze nem minden regiszterre igaz (elég, ha csak egyszer tévedünk), de a kísérletezés nem ajánlatos, főképp az elektronsugár-vezérlő (CRTC) regiszterek esetében nem!
2. Ha egy regiszterben redundáns (nem definiált jelentésű) bit található, akkor azt érdemes minden esetben maszkolni. Mivel több különböző típusú kártya van forgalomban, ezért előfordulhat, hogy az itt leírt regisztereknek egyéb funkciójuk is van (pl. az SVGA kártyákon). Másrészt a VGA kártyákat továbbfejlesztik, és lehet, hogy az eddig még definiálatlan helyekre új funkciók kerülnek majd. Tehát ha egy regisztert írunk, akkor ügyeljünk arra, hogy a redundáns helyekre mindig 0 érték kerüljön!

3.1. A VGA latch

A regiszterek ismertetése előtt beszélnünk kell a kártya belső tárolóeleméről, a VGA latch-ről (a latch átmeneti tárolót jelent). A 16 színű grafikus képfelépítésnél leírtuk, hogy egy képbájt 4 bitplane bájt együttese, ez az úgynevezett **színes bájt**. A latch is egy ilyen négybájtos tárolóelem, amely a CPU és a kártya közötti adatcserét valósítja meg.



11. ábra A latch szerkezete

Ha a CPU a képmemóriából adatot olvas, akkor ez a latch feltöltődését eredményezi. Nyilvánvaló, hogy a (CPU által) beolvasott adat csak a latch egyetlen bájtja lehet, vagyis egyszerre nem lehet az egész latch értékét megállapítani. Hogy az olvasás pontosan hogyan zajlik le, azt majd az olvasási módoknál ismertetjük. Képmemória-

íráskor is szerepet kap a latch, ugyanis a CPU adat és a latch adat valamilyen logikai kapcsolatba kerül, és a logikai művelet eredménye jut a képre. Ezt is meglátjuk majd az írásmódok leírásánál.

3.2. Az Attribute Controller regiszterek

Az Attribute Controller regiszterek, vagyis a működést, a kártya viselkedését vezérlő regiszterek a \$03C0(W) és a \$03C1(R) porton keresztül érhetők el. A regiszter cím- és adatregiszter is egyben, ezért az elérése egy különleges műveletsort igényel. A kártyán található egy belső tárolóelem (flip-flop), amely emlékszik arra, hogy a \$3C0 című port adat- vagy címmódban volt-e. Ezt a flip-flopot kell törölnünk, hogy a regiszter címmódba kapcsoljon. A flip-flop törlése a \$03DA című port olvasásával lehetséges. A regiszter címe melletti betű a regiszter lehetséges elérési módját mutatja. A **W** betű írást jelent, a regiszter csak írható, illetve írás esetén lesz a leírt a funkciója. Az **R** betű olvasást jelent, a regiszter csak olvasható, illetve a leírt funkció a regiszter olvasása esetére igaz. Az **R/W** jelölés azt jelenti, hogy a regiszter írható és olvasható is, funkciója a két esetben megegyezik.

\$03C0 címregiszter:

bit:

- 0—4: Az adatregiszter címe. A legközelebb kiírt adat abba a regiszterbe kerül, amelynek a címét ezen az öt biten megadtuk.
- 5: Beállított állapota engedélyezi a képet, és tiltja a palettaregiszterek módosítását.

\$03C0 index \$00..\$0F (R/W): adatregiszter: elsődleges(EGA) palettaregiszterek:

bit:

- 0—5: 6- (vagy 4-) bites index a 256 elemű RGB regiszterekre.

A regiszter használatára nézzünk egy példát! A rövid rutin beállítja a 0. palettaregisztert, azaz a háttérszínt a 05-ös RGB regiszterre:

```

Asm
mov    dx,03DAh      {Olvasás a 03DA portról:}
in     al,dx         {A 03C0 regisztert címmódba kapcsolja}
mov    dx,03c0h
mov    al,00h        {A 00-s palettaregiszter megcímzése}
out    dx,al
mov    al,05h        {05 kivitele a regiszterbe}
out    dx,al
mov    al,20h        {A képernyő engedélyezése (5. bit=1)}
out    dx,al

```

End;

Látható, hogy a regisztert mindvégig a \$03C0 porton értük el, és hogy minden írás invertálta a flip-flopot (cím, adat, cím...). Minden attribute regisztert ezzel a módszerrel lehet írni (a \$03C0 porton), illetve olvasni (a \$03C1 porton).

\$03C0 index \$10 (R/W): mód kontroll regiszter (Mode Control)

bit:

- 0: 1 = grafikus mód, 0 = karakteres mód.
- 1: Monokrom mód, ha 1, színes kép, ha 0.
- 2: A 9-pixeles karakterek 9. pixelének állapota. A karakterkészlet \$C0—\$DF kódú elemei a kilencedik pixelen a nyolcadik pixelt másolják (ezek a keretező karakterek), míg az összes többi karakter esetében a kilencedik pixel mindig háttér színű. Ez a bit a \$C0—\$DF karakterek 9. pixeljének az állapotát adja meg. Ha 1, akkor a 9. pixel a 8. pixel másolata, ha 0, akkor ezek a pixelek is háttér színűek.
- 3: Ha 1, akkor a karakteres kép színbájtjai „8 háttérszín és villogás” módban vannak, vagyis a színbájt 7. bitjének 1-es állapota a karakter villogását eredményezi. Ha ez a bit 0, akkor a színbájtok „16 háttér- és 16 tintaszín” módban vannak, vagyis a színbájt 7. bitjének 1-es állapota nagyobb fényerejű háttérszín eredményez.
- 4: Nem definiált.
- 5: Osztott képernyős üzemmódban (lásd később) az alsó kép eltolásának állapotát határozza meg. Ha 0, akkor az alsó kép horizontális eltolása követi a felső képet, ha 1, akkor az alsó kép nem mozdul.
- 6: Ha 1, akkor a pixelek 8-bitesek. A 256 színű mód használja.
- 7: Ha 1, akkor az RGB regiszterekre mutató palettaregiszterek 4—5. bitjét a \$3C0,\$14 regiszter 0—1. bitjei adják. A bit az EGA paletta és az RGB regiszterek viszonyát határozza meg. A viszony kétféle lehet:

- A palettaregiszterek 6-bitesek, így a 256 RGB regisztert 4 db 64 elemű tömbként lehet elérni.
- A palettaregisztereknek csak 4 bitjét használjuk, így a \$03C0, \$14 regiszterrel együtt 16 db 16 elemű tömbre osztva lehet használni a palettát. A kétféle felosztásnak 256 színű módban nincs értelme. Bővebben lásd a \$3C0 index \$14 regiszternél.

\$03C0 index \$11 (R/W): keret színregiszter (Overscan Color)

bit:

- 0—5: A keret színét meghatározó palettaregiszter száma.
- 6—7: Nem definiált.

\$03C0 index \$12 (R/W): bitplane-engedélyező regiszter (Plane Enable)

bit:

- 0: A 0. bitplane engedélyezése.
- 1: Az 1. bitplane engedélyezése.
- 2: A 2. bitplane engedélyezése.
- 3: A 3. bitplane engedélyezése.

4—5: Video státus, csak diagnosztikára használják.

6—7: Nem definiált.

\$03C0 index \$13 (R/W): horizontális eltolásregiszter (PEL Panning)

bit:

0—3: A képernyő balra tolását adja pixelben (vízszintes eltolás). A különböző videomódokban különbözőképpen kell értelmezni:

Érték	9-bites szöveges mód	256 színű mód	Más módok
0	1	0	0
1	2		1
2	3	1	2
3	4		3
4	5	2	4
5	6		5
6	7	3	6
7	8		7
8	0		

bit:

4—7: Nem definiált.

\$03C0 index \$14 (R/W): színtömbkiválasztó regiszter (Color Bank Select)

bit:

0—1: Ha a \$03C0, \$10 regiszter 7. bitje 1, akkor ez a két bit adja a palettaregiszterek 4—5. bitjeit.

2—3: A palettaregiszter 6—7. bitjei.

A \$03C0 index \$10 regiszter 7. bitjének állapotától függően így definiáljuk a 4/64 vagy a 16/16 palettatömböt. A regiszter értéke nem számít a 256 színű módban.

3.3. Státus- és kimeneti regiszterek

A regiszterek egyszerű IN/OUT utasításokkal közvetlenül elérhetők.

\$03C2 (R): input státus #0 regiszter

bit:

0—3: Nem definiált.

4: A \$03C2(W) regiszter 2—3. bitjének segítségével meghatározott kapcsoló állapotát adja meg. A kapcsoló állapota magas, ha 1. (Csak diagnosztikára használjuk.)

5—6: Nem definiált.

7: Vertikális interrupt kérés volt, ha 1. A bit létezése nem minden VGA esetén garantált!

\$03C2 (W): kimeneti állapot regiszter (Miscellaneous Output)

bit:

- 0: Ha 1, akkor a kártya színes módban van és a báziscím \$03Dx, ha 0, akkor a kártya üzemmódja monokrom és \$03Bx a báziscím.
- 1: Engedélyezi/tiltja a CPU hozzáférését a videomemóriához.
- 2—3: Horizontális órajel kiválasztás:
 - 00: 25 MHz (320/640, 8-pixeles karakterek).
 - 01: 28 MHz (360/720, 9-pixeles karakterek).
 - 10: Fejlesztésre fenntartva.
 - 11: Nem definiált.
- 4: Nem definiált.
- 5: Páros/páratlan módban a lapozást végzi.
- 6: Horizontális szinkron polaritás: 1=negatív, 0=pozitív.
- 7: Vertikális szinkron polaritás: 1=negatív, 0=pozitív.

A 6—7. bit kiválasztja a függőleges felbontást:

- 00: Fejlesztésre fenntartva.
- 01: 400 sor.
- 10: 350 sor.
- 11: 480 sor.

A regiszter értéke leolvasható a \$03CC portról.

\$03C3 (W): video alrendszer engedélyező regiszter (Video Subsystem Enable)

bit:

- 0: Engedélyezi a VGA képet, ha 1.
- 1—7: Nem definiált.

3.4. Video-sequencer regiszterek

A regiszterek írása két porton, a \$03C4 (cím) és a \$03C5 (adat) portokon történik. A \$03C4 portra kell kiírni a módosítani vagy olvasni kívánt regiszter számát (0..4), majd a \$03C5 című portról be lehet olvasni a megcímzett regiszter tartalmát, illetve ide kell írni a kívánt értéket.

\$03C4 index \$00 (R/W): Sequencer Reset regiszter

bit:

- 0—1: Szinkron/aszinkron reset indítása.
 - 01: Szinkron.
 - 10: Aszinkron.
 - 11: Normál állapot.
- 2—7: Nem definiált.

Figyelem!!!

A *Sequencer Reset* regiszter használatát nem minden VGA követeli meg, de bizonyos (főként régebbi típusú) kártyák esetén szükség lehet a *sequencer* regiszterkészlet inicializálására egy *sequencer szinkron reset* segítségével. A következő \$3C4, \$01 órajelmód regiszter minden írása hibához vezethet néhány VGA esetében (pl. OAK). Ezért minden egyes példaprogramban, amelyet a későbbiekben bemutatunk, a következő feltételes fordítási direktíva szerepel:

```

        mov     dx,03c4h
{ $IFDEF SeqRESET }
        mov     ax,0100h      { Szinkron reset (a kép eltűnik)          }
        out     dx,ax
{ $ENDIF }

        .. sequencer órajel mód módosítás ...

{ $IFDEF SeqRESET }
        mov     ax,0300h      { Normál állapot (a kép ismét látható)    }
        out     dx,ax
{ $ENDIF }

```

A fenti feltételes fordítás minden egyes példaprogramban benne lesz (ahol szükséges), de a könyv további részében erre már nem térünk ki! Ezért minden programban definiáltuk a **SeqRESET** szimbólumot. Hogy egy adott VGA kártya igényli-e a reset metódust, azt minden egyes esetben ki kell próbálni. Ha a program a szimbólum eltávolítása után is jól működik, akkor azon a kártyán nincs szükség a reset metódusra (pl. Paradise, Trident, Realtek).

\$03C4 index \$01 (R/W): órajelmód regiszter (Clocking Mode)

bit:

- 0: Ha 1, akkor 8-pixeles karakter órajel, ha 0, akkor 9-pixeles.
- 1: Nem definiált.
- 2: A videoadatok letöltődését határozza meg: 1: minden második ciklusnál, 0: minden egyes ciklusnál. (CPU/videociklus megosztás.)
- 3: Pixel órajel: ha 1, akkor a pixelek dupla szélesek. Ez a bit kapcsol át a 320-as és a 640-es mód között. (Ennek a bitnek az átállítása nem elegendő a 640/320-as szélesség átváltásához!)
- 4: Mint a 2. bit, de ez itt minden negyedik ciklust határozza meg.
- 5: Ha 1, akkor kikapcsolja a képernyőt, és minden memóriaciklus a CPU-é lesz.
- 6—7: Nem definiált.

\$03C4 index \$02 (R/W): bitplane írásaszk regiszter (Plane Write Enable)

bit:

- 0: Engedélyezi a 0. bitplane írását.
- 1: Engedélyezi az 1. bitplane írását.
- 2: Engedélyezi a 2. bitplane írását.
- 3: Engedélyezi a 3. bitplane írását.

\$03C4 index \$03 (R/W): karakter bittérkép kiválasztás (Char Bank Select)

bit:

- 0,1,4: A karakterkészlet sorszámát (0..7) határozza meg, ha a színbájt 3. bitje 0.
- 2,3,5: A karakterkészlet sorszámát (0..7) határozza meg, ha a színbájt 3. bitje 1.
- 6—7: Nem definiált.

A különböző sorszámú karakterkészletek az A000:xxxx címtől a következő távolságra helyezkednek el a memóriában:

- | | |
|--------|--------|
| 0: 0K | 1: 16K |
| 2: 32K | 3: 48K |
| 4: 8K | 5: 24K |
| 6: 40K | 7: 56K |

Látható, hogy a VGA kártya 8 db különböző karakterkészlet közül tud választani, és egyszerre 512 karaktert tud megjeleníteni. Mivel a karakterkód csak 8-bites, ezért a karakter kódjának 9. bitjét a színbájt 3. bitje adja. Alapesetben mindkét készlet azonos, azaz csak 256 karaktert tudunk kezelni.

\$03C4 index \$04 (R/W): memóriamód regiszter (Memory Mode)

bit:

- 0: 1 szöveges módban, 0 grafikus módban.
- 1: 1, ha a video-RAM nagyobb 64 Kbájt nál.
- 2: Engedélyezi a CGA kompatibilis páros/páratlan videocímzést.
- 3: Ha 1, akkor a videocím 0—1. bitjei választják ki az aktuális bitplane-t, ha 0, akkor ezt a \$03C4, \$02 és a \$03CE, \$04 regiszterek végzik. Ez a kombinált videocímzés a 256 színű mód programozásánál kap szerepet, ott majd részletesen ismertetni fogjuk.
- 4—7: Nem definiált.

3.5. RGB regiszterek

Az RGB regiszterek a VGA RGB palettájának elérését teszik lehetővé. A belső 256*3 bájtos színtáblázat egyenként vagy blokkokban írható és olvasható. A színregiszterek komponensei (R, G és B) 6-bitesek.

\$03C6 (R/W): RGB maszkregiszter: (RGB mask)

bit:

0—7: RGB maszk, belső használatra, mindig \$FF-re van állítva. Nem kell módosítani!

\$03C7 (R): RGB állapotregiszter: (DAC State)

bit:

0—1: Az RGB regiszterek hozzáférési módját adja meg:

00: Olvasás.

11: Írás.

2—7: Nem definiált.

\$03C7 (W): RGB olvasás címregiszter (RGB Read Address)

bit:

0—7: Az olvasni kívánt RGB regiszter száma. A \$03C7 port írása után a \$03C9 port háromszori olvasása megadja a regiszter R, G és B összetevőjét, és megnöveli a címregisztert.

A következő program a 34. RGB regisztert kérdezi le:

Asm

```

mov    dx, 03C7h
mov    al, 34
out    dx, al           {A 34. RGB regiszter megcímezése }
mov    dx, 03C9h
in     al, dx
mov    cl, al           {CL = R komponens }
in     al, dx
mov    ch, al           {CH = G komponens }
in     al, dx
mov    dl, al           {DL = B komponens }
```

End;

\$03C8 (R/W): RGB írás címregiszter (RGB Write Address)

bit:

0—7: Az írni kívánt RGB regiszter száma. A \$03C8 port írása után a \$03C9-re kivitt három bájt definiálja az RGB komponenseket, és megnöveli a címmutatót, azaz a következő RGB regiszterre áll.

A következő program pirosra állítja a 0. RGB regisztert:

Asm

```

mov    dx,03C8h
xor     al,al
out     dx,al          {A 0. RGB regiszter megcímezése    }
mov     dx,03C9h
mov     al,63          { Az R komponens 63 (maximum)      }
out     dx,al
xor     al,al          { AL = 00                          }
out     dx,al          { A G komponens 0                    }
out     dx,al          { A B komponens 0                    }
```

End;

\$03C9 (R/W): RGB adatregiszter (RGB Data)

bit:

0—5: A következő RGB komponens értéke (0..63).

6—7: Nem definiált.

A regisztert a \$03C7 és \$03C8 regiszterrel együtt használjuk.

\$03CA (R): vertikális szinkron olvasás (Feature Control)

bit:

0—2: Nem definiált.

3: Az éppen beállított vertikális szinkron.

4—7: Nem definiált.

A regiszter a \$03DA regiszter beállított értékét kérdezi le.

\$03CC (R): a kimeneti állapotregiszter státusa (Miscellaneous Output)

A regiszter a \$03C2 regiszter értékét kérdezi le (lásd ott).

3.6. Grafikai vezérlő regiszterek

\$03CE : Címregiszter

\$03CF: Adatregiszter

E regiszterek írása ugyanúgy történik, mint a *sequencer* regisztereké: először a \$03CE portra kiírjuk a regiszter számát (0..8), majd a \$03CF portra az adatot.

\$03CE index \$00 (R/W): írás vagy törlés (Set/Reset) regiszter

bit:

- 0: A 0. bitplane írása/törlése, ha a \$3CE, \$01 regiszter engedi.
- 1: Az 1. bitplane írása/törlése, ha a \$3CE, \$01 regiszter engedi.
- 2: A 2. bitplane írása/törlése, ha a \$3CE, \$01 regiszter engedi.
- 3: A 3. bitplane írása/törlése, ha a \$3CE, \$01 regiszter engedi.
- 4—7: Nem definiált.

\$03CE index \$01 (R/W): írás/törlés engedélyezése (Enable Set/Reset) regiszter

bit:

- 0: Engedélyezi a \$3CE, \$00 regisztert írásra/törlésre (0. plane).
- 1: Engedélyezi a \$3CE, \$00 regisztert írásra/törlésre (1. plane).
- 2: Engedélyezi a \$3CE, \$00 regisztert írásra/törlésre (2. plane).
- 3: Engedélyezi a \$3CE, \$00 regisztert írásra/törlésre (3. plane).
- 4—7: Nem definiált.

A \$3CE, \$00 és \$01 regiszterek a **Mode 0** és **Mode 3** írás módban kapnak szerepet (lásd ott).

\$03CE index \$02 (R/W): szín-összehasonlító regiszter (Color Compare)

bit:

- 0—3: Az összehasonlítási szín.
- 4—7: Nem definiált.

A regiszter szín-összehasonlító olvasás módban a keresett színt adja meg. (Bővebben az olvasási módoknál, **Mode 1** esetben.)

\$03CE index \$03 (R/W): adatelforgatás regiszter (Data Rotate & Fn. Select)

bit:

- 0—2: Az elforgatás száma, mielőtt az adat kikerülne a képernyőre (**Mode 0** és **Mode 3** írás módban).
- 3—4: A CPU adat és a VGA latch közötti logikai műveletet adja meg 0. és 2. írás módban:
 - 00: A CPU adat nem módosul.
 - 01: A CPU adat ÉS (AND) kapcsolatba kerül a latch adattal.
 - 10: A CPU adat VAGY (OR) kapcsolatba kerül a latch adattal.
 - 11: A CPU adat KIZÁRÓ VAGY (XOR) kapcsolatba kerül a latch adattal.
- 5—7: Nem definiált.

\$03CE index \$04 (R/W): bitplane kiválasztása olvasáskor (Read Plane Select)

bit:

- 0—1: A két bit kiválasztja az olvasni kívánt bitplane számát a 0. olvasás módban. A CPU-olvasás művelete során a teljes VGA latch feltöltődik, de ez a regiszter meghatározza, hogy a latch melyik bájtja adódik át a CPU-nak.
- 2—7: Nem definiált.

\$03CE index \$05 (R/W): írás és olvasás mód regiszter (Mode)

bit:

0—1: írás mód beállítása

00: 0. írás mód. Ez az írás mód olvasás—módosítás—írás operációval dolgozik. Először beolvasunk egy bájtot a videomemóriából, ekkor a latch feltöltődik. Ez után kiírjuk a kívánt adatot a videomemóriába, a CPU adat logikai kapcsolatba kerül a latch-csel (\$03CE,\$03), és elfordul a megfelelő bitszámmal. Az írni kívánt bitplane-eket a \$03C4, \$02 regiszterrel lehet engedélyezni és tiltani.

01: 1. írás mód. Ez a mód a videomemória-videomemória átvitelt valósítja meg. A CPU beolvas egy adatot a videomemóriából, ekkor a latch feltöltődik, majd kiírja ezt az adatot a videomemóriába. Így a teljes latch átmásolódik, azaz egy egybájtos átviteli utasítással lehet 4 bájtot (minden plane-t) mozgatni. Ebben az írás módban egy egyszerű MOVSB utasítással képesek vagyunk minden pixelt átmozgatni a forrásból a célba. A könyv további részében az ilyen bájtokat „színes bájt”-oknak fogjuk hívni.

10: 2. írás mód. Ebben a módban egy színt tudunk írni a célcím kiválasztott biteibe. A CPU először beolvas egy adatot a képről (feltöltődik a latch). Ez után a \$03CE, \$08 regiszterrel kiválasztjuk a módosítani kívánt biteket, majd az adott címre kiírjuk a megfelelő színt. Ahol a \$03CE, \$08 regiszterben 1-es volt, oda a megadott szín fog kerülni, ahol pedig 0, oda a latch eredeti értéke.

11: 3. írás mód. A 0. és a 2. mód kombinációja.

2: Nem definiált.

3: Olvasás mód:

0: 0. olvasás mód. Az adat valamelyik bitplane-ről kerül a CPU-ba. Az olvasandó bitplane számát a \$03CE, \$04-es regiszterrel lehet beállítani.

1: 1. olvasás mód: szín-összehasonlítás. A CPU által megkapott bájt azon bitjei lesznek 1-esek, ahol a beolvasott adat (latch) színe egyezik a \$03CE, \$02 regiszterbe írt értékkel. Az összehasonlítást befolyásolja a \$03CE, \$07 regiszter (lásd ott).

4: Engedélyezi a páros/páratlan módot (CGA), ha 1.

5: Engedélyezi a CGA 2 bit/pixel módot (320×200 4 szín).

6: Engedélyezi a 256 színű módot.

7: Nem definiált.

Az írás és olvasás módok részletes magyarázatára a fejezet végén térünk ki.

\$3CE index \$06 (R/W): memórialap regiszter (Miscellaneous)

bit:

0: Grafikus mód, ha 1, szöveges mód, ha 0.

1: Páros/páratlan mód engedélyezése.

2—3: A memórialap címének kiválasztása:

00: A000:0000—B000:FFFF 128K videomemória

01: A000:0000—A000:FFFF VGA grafikus módok

10: B000:0000—B000:07FF Monokrom mód (Hercules)

11: B800:0000—B000:FFFF Szöveges módok

\$03CE index \$07 (R/W): szín-összehasonlítás maszkregiszter (Color Don't Care)

bit:

- 0: Kizárja a 0. bitplane-t az összehasonlításból, ha 0.
- 1: Kizárja az 1. bitplane-t az összehasonlításból, ha 0.
- 2: Kizárja a 2. bitplane-t az összehasonlításból, ha 0.
- 3: Kizárja a 3. bitplane-t az összehasonlításból, ha 0.
- 4—7: Nem definiált.

A regiszter az 1. olvasás módban érvényes.

\$03CE index \$08 (R/W): bitmaszk regiszter (Bit Mask)

bit:

- 0—7: A CPU adatot maszkolja ki, mielőtt az kiíródna a memóriába. Ahol 1-esek vannak, oda a CPU adatbitjei kerülnek, ahol 0-k vannak, oda pedig a latch bitjei.

3.7. Elektronsugár-vezérlő regiszterek (CRTC)

\$03D4: Címregiszter

\$03D5: Adatregiszter

Ezek a regiszterek végzik az elektronsugár pozicionálását a képen, valamint néhány memóriacím kezelését is. A \$00..\$09 és \$15, \$16, \$17 regiszterekkel nem érdemes kísérletezgetni, mert egy nem egészen jól sikerült beállítás megsértheti a hardvert! A beállítás módjára ugyanaz érvényes, mint a *sequencer* regiszterekre.

\$03D4 index \$00 (R/W): Horizontal Total regiszter

bit:

- 0—7: A teljes vízszintes karakterciklusok száma–5.

\$03D4 index \$01 (R/W): Horizontal Display End regiszter

bit:

- 0—7: A képen lévő karakterek (ciklusok) száma–1.

\$03D4 index \$02 (R/W): Start Horizontal Blanking regiszter

bit:

- 0—7: A vízszintes üres rész kezdőkarakterének száma.

\$03D4 index \$03 (R/W): End Horizontal Blanking regiszter

bit:

- 0—4: A vízszintes üres rész véget ér, ha a karakterszámláló alsó 6 bitje egyenlő ezzel az értékkel. Az 5. bit a \$03D4, \$05-ös regiszterben van.
- 5—6: A várakozási karakterciklusok száma, miután a karakterszámláló elérte a \$3D4, \$00-t.
- 7: Hozzáférhetővé teszi a *Vertical Retrace* regisztert, ha 1.

\$03D4 index \$04 (R/W): Start Horizontal Retrace regiszter

bit:

- 0—7: A vízszintes visszafutás megkezdődik, ha a karakterszámláló eléri ezt az értéket.

\$03D4 index \$05 (R/W): End Horizontal Retrace regiszter

bit:

- 0—4: A vízszintes visszafutás véget ér, ha a karakterszámláló alsó 5 bitje egyenlő ezzel az értékkel.
- 5—6: A várakozási karakterciklusok száma a vízszintes visszafutás után.
- 7: Az *End Horizontal Blanking* regiszter 5. bitje.

\$03D4 index \$06 (R/W): Vertical Total regiszter

bit:

- 0—7: A teljes függőleges rastersorok számának alsó 8 bitje. A rastersorok számának 8. és 9. bitje a \$03D4, \$07 regiszterben van.

\$03D4 index \$07 (R/W): túlcscordulás regiszter (Overflow)

bit

- 0: A *Vertical Total* regiszter (\$03D4, \$06) 8. bitje.
- 1: A *Vertical Display End* regiszter (\$03D4, \$12) 8. bitje.
- 2: A *Vertical Retrace Start* regiszter (\$03D4, \$10) 8. bitje.
- 3: A *Start Vertical Blanking* regiszter (\$03D4, \$15) 8. bitje.
- 4: A *Line Compare* regiszter (\$03D4, \$18) 8. bitje.
- 5: A *Vertical Total* regiszter (\$03D4, \$06) 9. bitje.
- 6: A *Vertical Display End* regiszter (\$03D4, \$12) 9. bitje.
- 7: A *Vertical Retrace Start* regiszter (\$03D4, \$10) 9. bitje.

\$03D4 index \$08 (R/W): Preset Row Scan regiszter, függőleges eltolás

bit:

- 0—4: A függőleges eltolás egy karakteren belül. Ezzel a függőleges görgetést (SCROLL) lehet megvalósítani karakteres kép esetén.
- 5—6: A balra eltolt bájtok száma a legelső rastersorban. Ez a kép vízszintes eltolása esetén használatos.
- 7: Nem definiált

\$03D4 index \$09 (R/W): Maximum Scan Line regiszter, karaktermagasság

bit:

- 0—4: Az egy karakterhez tartozó rastersorok száma–1. Ezek a bitek definiálják a karakterek magasságát. A magasság 0..31 között állítható.
- 5: A *Start Vertical Blanking* regiszter 9. bitje.
- 6: A *Line Compare* regiszter 9. bitje.
- 7: Ha 1, akkor minden rastersor megduplázódik. (Pl. 400 soros képen így lehet 200 soros felbontást beállítani.)

\$03D4 index \$0A (R/W): Cursor Start regiszter, a kurzor tetejének sorszáma a karakteren belül

bit:

- 0—4: A kurzor felső rastersorának száma.
- 5: Kikapcsolja a kurzort, ha 1.
- 6—7: Nem definiált.

\$03D4 index \$0B (R/W): Cursor End regiszter, a kurzor aljának sorszáma a karakteren belül

bit:

- 0—4: A kurzor alsó rastersorának száma.
- 5—6: A kurzor várakozása karakterciklusokban.
- 7: Nem definiált.

A \$03D4, \$0A és \$0B regiszterekkel a kurzor alakját adhatjuk meg.

\$03D4 index \$0C (R/W): Start Address High regiszter, a képernyő kezdőcíme

bit:

- 0—7: A képernyő kezdőcímének felső bájtja.

\$03D4 index \$0D (R/W): Start Address Low regiszter, a képernyő kezdőcíme

bit:

- 0—7: A képernyő kezdőcímének alsó bájtja.

\$03D4 index \$0E (R/W): Cursor Position High regiszter, a kurzor címe

bit:

- 0—7: A kurzor pozíciójának, pontosabban a kurzor „alatti” karakter címének felső bájtja.

\$03D4 index \$0F (R/W): Cursor Position Low regiszter, a kurzor címe

bit:

- 0—7: A kurzor címének alsó bájtja

\$03D4 index \$10 (R/W): Vertical Retrace Start regiszter, függőleges visszafutás

bit:

- 0—7: A függőleges visszafutás regiszter alsó 8 bitje. A további bitek a \$03D4, \$07 regiszterben találhatók. A vertikális visszafutás akkor kezdődik el, amikor a rastersorszámoló eléri ezt az értéket.

\$03D4 index \$11 (R/W): Vertical Retrace End regiszter, vízszintes visszafutás

bit:

- 0—3: A vertikális visszafutás akkor ér véget, amikor a rastersorszámoló alsó 4 bitje megegyezik ezzel az értékkel.
- 4: Ha 0, akkor nincs megszakítás a vertikális visszafutáskor.
- 5: Tiltja a *Vertical Blank* megszakítás kiadását (IRQ 2).
- 6: Ha 1, akkor 1 rastersorhoz 5 frissítési ciklus tartozik, ha 0, akkor 3.
- 7: Ha 1, akkor a \$03D4, \$00—\$07 regiszterekbe nem lehet írni!

\$03D4 index \$12 (R/W): Vertical Display End regiszter, függőleges képméret

bit:

- 0—7: A függőleges képhossz alsó 8 bitje. A látható kép véget ér, ha a rastersor-számláló eléri az itt lévő értéket. A regiszter további bitjei a \$03D4, \$07 regiszterben találhatók.

\$03D4 index \$13 (R/W): Offset regiszter (Logical Size), a sor vízszintes mérete

bit:

- 0—7: Az egy vízszintes sorban lévő bájtok számát adja meg.

Bájtok száma=\$03D4, \$13/K, ahol K:

- 2 bájt címezésmódnál (*byte*)
- 4 szó címezésmódnál (*word*)
- 8 dupla szó címezésmódnál (*dword*).

\$03D4 index \$14 (R/W): Underline Location regiszter, az aláhúzás helye

bit:

- 0—4: Megadja a karakterboxon belül az aláhúzás pozícióját.
- 5: Ha 1, akkor a memóriacímek csak minden 4. karakterciklusban változnak (vízszintes torzítás).
- 6: Dupla szó címezésmód, ha 1.
- 7: Nem definiált.

\$03D4 index \$15 (R/W): Start Vertical Blank regiszter, a függőleges üres rész kezdete

bit:

- 0—7: A *Start Vertical Blank* regiszter alsó 8 bitje. A további bitek a \$03D4, \$07 regiszterben találhatók. A függőleges üres rész akkor kezdődik, amikor a rastersorszámoló eléri ezt az értéket.

\$03D4 index \$16 (R/W): End Vertical Blanking regiszter, a függőleges üres rész vége

bit:

- 0—6: A függőleges üres rész véget ér, ha a rastersorszámoló alsó 7 bitje egyenlő ezzel az értékkel.

\$03D4 index \$17 (R/W): Mode Control regiszter, üzemmód beállítása

bit:

- 0: Ha 0, akkor a kártya CGA kompatibilis memóriacímzést használ.
- 1: Ha 0, akkor a kártya a Hercules kártyával kompatibilis memóriacímzést használ.

- 2: Ha 1, akkor a rastersorszámoló csak minden második sornál növekszik.
- 3: Ha 1, akkor a memóriacím csak minden második karakterciklusban növekszik.
- 4: Fenntartva az SVGA chipek számára.
- 5: ??? Nincs pontosan definiált szerepe. (Ha ez a bit 1, akkor szó címzés módban a 15. bit a 0. bitbe tolódik forgatás esetén; ha a bit 0, akkor a 13. bit kerül a 0. bitpozícióba. Nem minden VGA esetén működik!)
- 6: ??? Nincs pontosan definiált szerepe. (Törölt állapota jelzi a szó címzés módot. Nem minden VGA esetén garantált a működése!)
- 7: Ha 0-ba állítjuk, akkor törli a videorendszert, és amíg ismét 1-be állítjuk, a kép nem látható (nem garantált a hatása).

\$03D4 index \$18 (R/W): Line Compare regiszter, a sorszámoló összehasonlítása
bit:

- 0—7: A *Line Compare* alsó 8 bitje. Amikor a rastersorszámoló eléri az itt beállított sornak a számát, a videokezdőcím (\$03D4, \$0C, \$0D) 0-ba áll. A regiszterrel osztott képernyős módot lehet beállítani. A regiszter tízbites, a hiányzó két bit a \$3D4, \$07 regiszter 4. bitjében és a \$3D4, \$09 regiszter 6. bitjében található.

3.8. Egyéb regiszterek

\$03DA (R): státuszregiszter

bit:

- 0: Vertikális vagy horizontális visszafutás van, ha 1.
- 1—2: Nem definiált.
- 3: Vertikális visszafutás van, ha 1.
- 4—7: Nem definiált.

\$03DA(W): Vertikális szinkron vezérlés

bit:

- 0—2: Nem definiált.
- 3: A vertikális szinkron kiválasztása. (Belső használatra fenntartott funkció.)

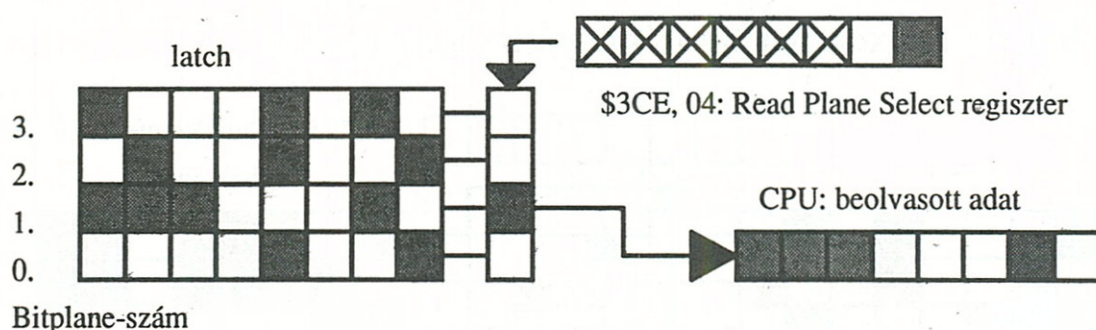
Ezzel végigvettük a VGA kártya regisztereit. Bizonyára sokaknak feltűnik, hogy egyes bitek sok helyen ismétlődnek, vagy több bitnek azonos a funkciója. A VGA-BIOS azért jött létre, hogy ne legyen szükség a regiszterek közvetlen programozására. A példákban látni fogjuk, hogy amit csak lehet, azt a BIOS-ra bízunk, de lesz néhány olyan probléma, amelyet csak közvetlen programozással tudunk majd megoldani. A felsorolt regiszterek között vannak számunkra lényegtelenek is, de vannak olyanok is, amelyeket gyakran fogunk használni.

3.9. A képmemória olvasása és írása

3.9.1. Olvasási módok

A VGA kétféle olvasás módot ismer, amelyekkel megkaphatjuk a képernyőn lévő adatokat. Az olvasás módoknak leginkább a bitplane technikájú grafikus videomódoknál van szerepe.

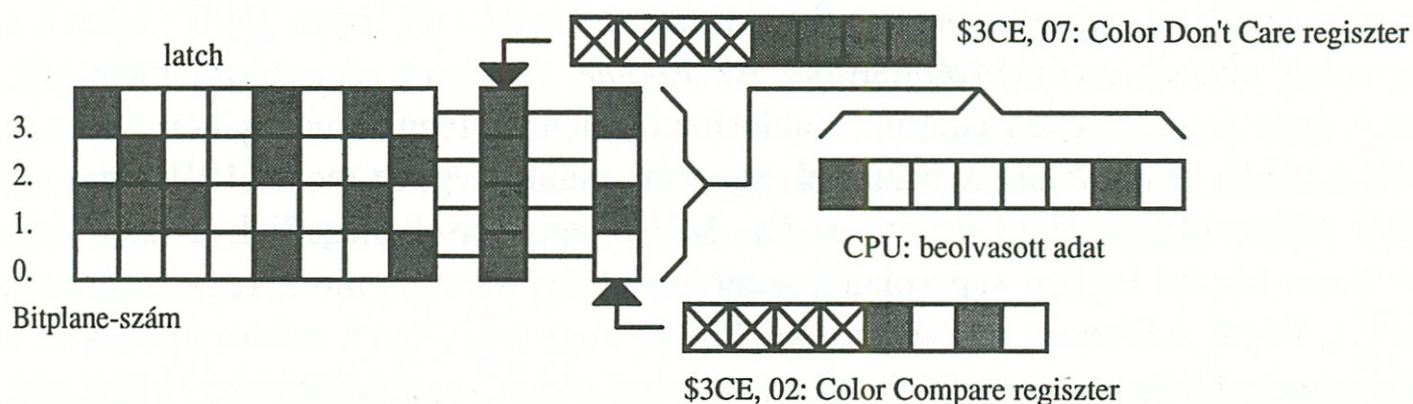
Mode 0: Egyszerű bitplane-leolvasás. Ezzel a móddal a négy bitplane valamelyikét tudjuk leolvasni.



12. ábra A Mode 0 olvasás mód

A \$3CE, \$04 Read Plane Select regiszter alsó két bitjével meg kell adni az olvasandó bitplane számát (0, 1, 2 vagy 3), majd egy egyszerű olvasást kell végezni a videomemóriából. Az olvasás hatására feltöltődik a latch, és a megfelelő bájtja a CPU-hoz kerül.

Mode 1: Szín-összehasonlító olvasás. Az olvasással meg tudjuk határozni, hogy a színes képbájt mely bithelyein van az általunk lekérdezett szín.



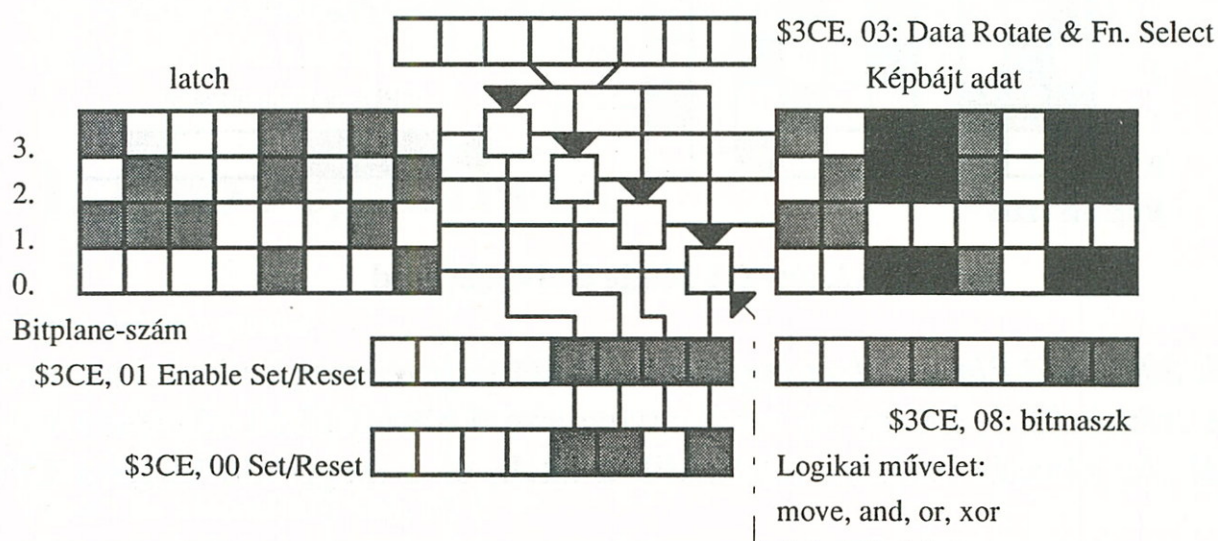
13. ábra A Mode 1 olvasás mód

A *Color Don't Care* regiszterrel meg tudjuk határozni, hogy melyik bitplane ne vegyen részt az összehasonlításban. A fenti példában ebben minden plane részt vesz. A *Color Compare* regiszterrel megadjuk a keresett színt (ebben a példában ez 1010, azaz 10, világoszöld), és az olvasás után a CPU által kapta bájt azon bitjei lesznek egyesek, ahol a megadott szín található.

3.9.2. Írás módok

Négyféle írás mód közül választhatunk. Az egyes írás módokból kiderül, hogy a VGA kártyát pixelorientált grafikus programozásra fejlesztették ki, és a legtöbb lehetőség ezt támogatja.

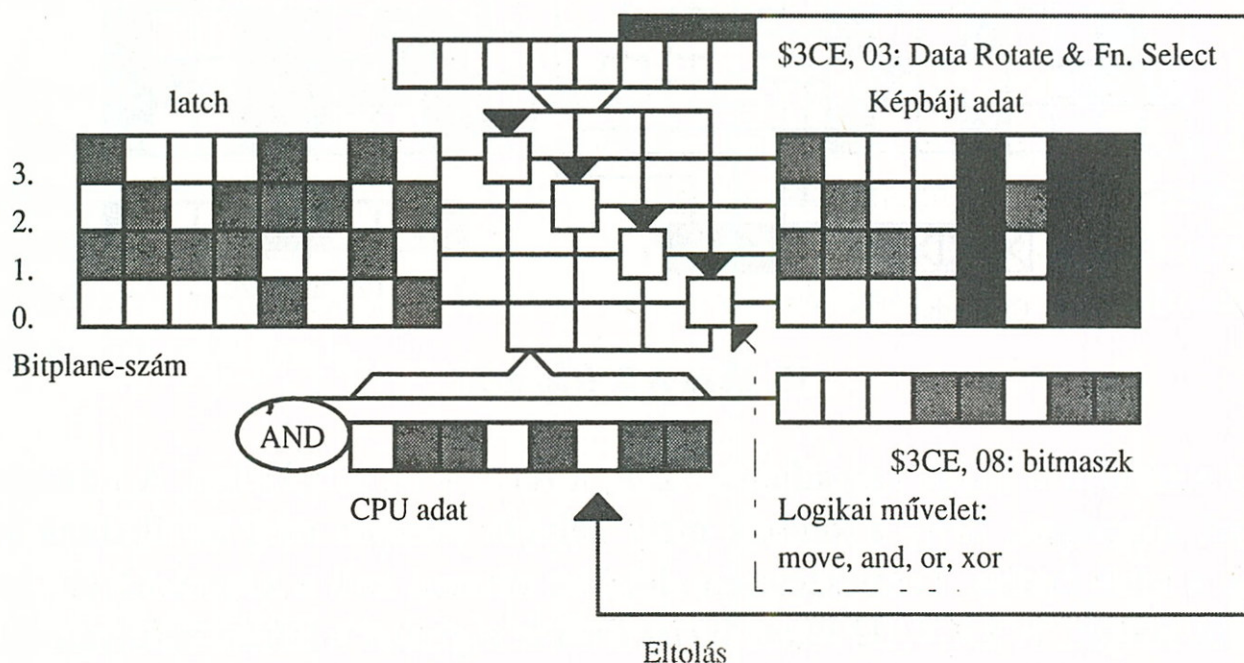
Mode 0: A bájt- és a pixelorientált írás együttes alkalmazása.



14.a ábra A 0/a. írás mód

Ebben az esetben a CPU által kiadott adatnak nincs jelentősége, mert a *\$3CE, 01 Enable Set/Reset* regiszter nem nulla. Az írás a következőképpen zajlik: a latch a legutolsó olvasás eredményét tárolja. Az *Enable Set/Reset* regiszterrel megadjuk, hogy mely bitplane-eken akarunk módosítani, jelen esetben mindegyiken, ezért az értéke 1111b. A *Set/Reset* regiszterrel egy színt adunk meg, ez most 1101b, azaz 13, tehát világoslila. A *Data Rotate & Fn. Select* regiszterrel megadjuk a latch és a *Set/Reset* közötti logikai kapcsolatot, amely most egy szimpla **move** (egyszerű mozgatás). Végül a *Bitmask* regiszterrel kijelöljük, hogy mely bitek módosuljanak, és a kártya mely biteket vegye a latch-ből. Az írás ezt a kombinált értéket viszi ki a képre. A *\$3C4,02 Plane Write Enable* regiszterrel az egész folyamatban ki tudjuk jelölni, hogy mely plane-ek vegyenek részt az írásban. A fenti példában e regiszter értéke 1111b, azaz minden plane írásra engedélyezett.

A második esetben az Enable Set/Reset értéke nulla, ezért most a CPU adat is módosítja a kiírást:

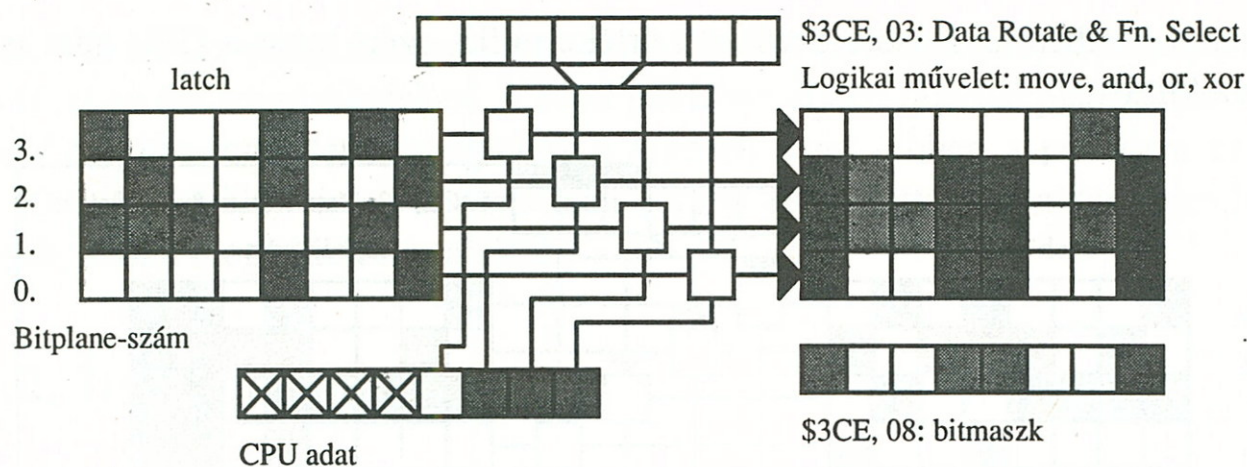


14.b ábra A 0/b. írás mód

Ebben az esetben a *bitmaszk* és a CPU adat logikai ÉS (AND) kapcsolatba kerül, így alakul ki a valódi kiírandó adat, amely most 00001011b. A *Data Rotate & Fn. Select* megadja a logikai műveletet (most szimpla **move**) és a CPU adatának elforgatását (ez most 0, tehát az adat nem tolódik el). A bitmaszk által megadott helyeken megjelenik az előzőleg képzett adat, a nulla helyeken pedig a latch-ből származó adat. A két eset között lényeges különbség, hogy az első esetben a *Set/Reset regiszter* szint tartalmazott, azaz a beírás „függőlegesen” (bitenként) történt, míg a második esetben a beírásra „vízszintesen” (bájtonként) került sor. A második esetben is érvényes a *\$3C4, 02 Plane Write Enable* regiszter hatása. Ha tehát ebbe például 0110b értéket írunk, akkor a 0. és a 3. plane-re a latch eredeti értéke kerül, mert tiltottuk az ide való írást, míg az 1. és a 2. plane-re a CPU adat kerül!

Mode 1: Bájtorientált mozgatás, másolás képmemóriából képmemóriába. Ez az írás mód a forrás színes bájtot egyszerűen átmásolja a cél színes bájtba. Ezzel a módszerrel minden plane tartalma egyszerre mozgatható egy egyszerű **movsb** utasítással. A latch 8-bites, ezért csak bájt méretű másolást tudunk megvalósítani. A **\$3C4, 02 Plane Write Enable** regiszter a kiírást befolyásolja.

Mode 2: Pixelorientált írás. Az általunk kiválasztott bithelyekre az általunk megadott szín kerül.

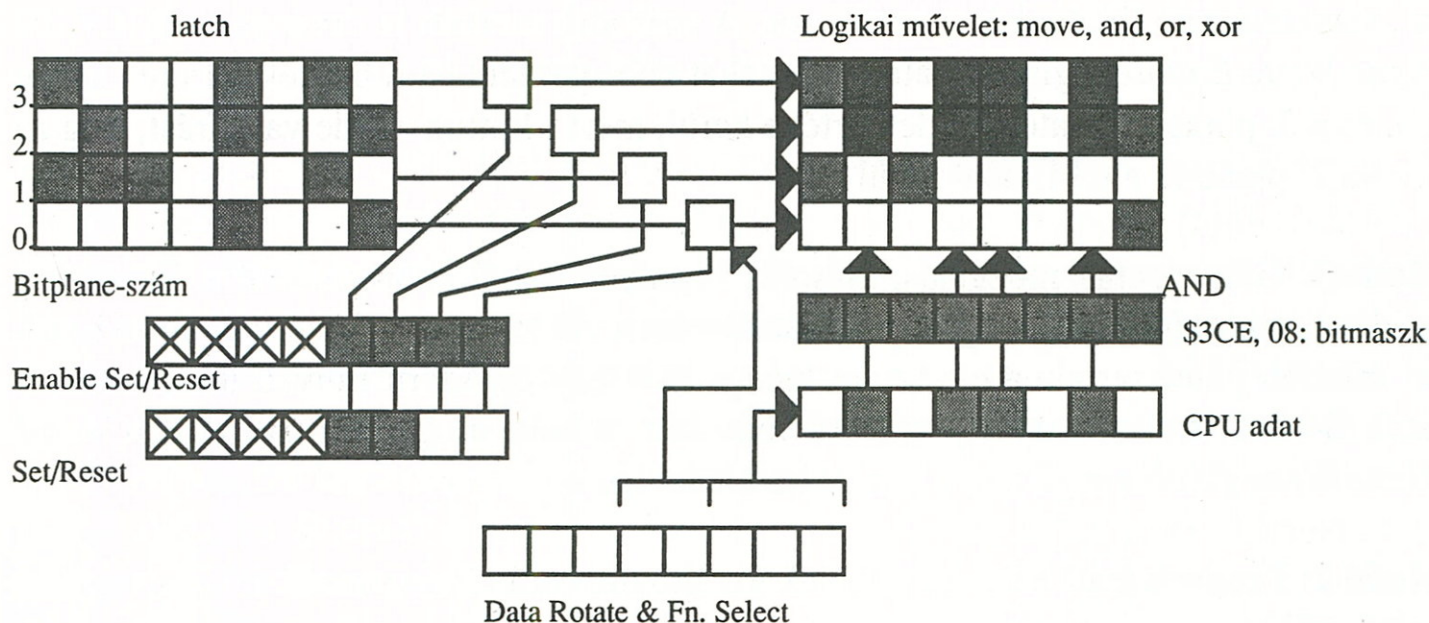


15. ábra A 2. írás mód

A CPU adat alsó 4 bitjén megadjuk a színt, a felső 4 bit érdektelen. A *bitmaszk* regiszterrel megadjuk, hogy a képbájt mely helyein jelenjen meg az általunk kívánt szín, és a latch értéke. Természetesen előzőleg olvasni kell a képmemóriát, hogy a latch feltöltődjön. A *\$3C4, 02 Plane Write Enable* regiszter a kiírást befolyásolja.

Mode 3: A 0. és a 2. mód kombinációja.

Ebben az esetben a *Set/Reset* és az *Enable Set/Reset* regiszterek logikai ÉS kapcsolata adja a színt, míg a *bitmaszk* és a CPU adat logikai ÉS kapcsolata a bithelyeket. A *Data Rotate & Fn. Select* megadja a CPU adat elforgatásainak számát (amely a példánkban nulla) és a logikai műveletet, ez most **move**. Ezen a módon egy meghatározott színt tudunk kivinni a meghatározott bithelyekre, de a CPU adat a **Mode 2**-től eltérően nem szín, hanem bitmaszkot tartalmaz.



16. ábra A 3. írás mód

Mindegyik írásmódra érvényes, hogy amennyiben még nem olvastunk a képről adatot, akkor a latch minden bitje nulla, ha pedig olvastunk, akkor a latch ezt a következő olvasásig megőrzi.

A lemezmellékleten található egy **VGAREGS.TXT** nevű szövegfájl, amely a regiszterek leírását tartalmazza rövidített formában. Talán nem haszontalan ezt a fájlt ki nyomtatni, hogy a későbbi példáknál ne kelljen minduntalan visszalapozni az egyes regiszterek bitkiosztása miatt. Mivel a regiszterek a továbbiakban a programozás alapját képezik, mindenképpen szükség van az alapos ismeretükre! Vonatkozik ez az olvasás és írás módokra is.

4. A VGA-BIOS szubrutinjai

A BIOS videóval kapcsolatos szolgáltatásait az **INT 10h** utasítással lehet elérni. A paraméterátadás más megszakításokhoz hasonlóan a processzor regiszterein keresztül történik, az esetleges visszaadott értékeket is a regiszterekben kapjuk meg. A leírt adatok az IBM VGA szabvány szerinti kártyákra vonatkoznak, más típusú VGA kártyák esetén apróbb eltérések előfordulhatnak.

SET VIDEO MODE, a videomód beállítása

AH 00h

AL a kívánt üzemmód, videomód száma

A VGA kártyán beállítható szabványos videomódok (a text szöveges, a graph pedig grafikus üzemmódot jelent):

Mód (AL)	Típus	Felbontás	Színek száma	Lapok száma	A kép kez- dőcíme
00h	text	40x25	16	8	b800:0000
01h	text	40x25	16	8	b800:0000
02h	text	80x25	16	4	b800:0000
03h	text	80x25	16	4	b800:0000
04h	graph	320x200	mono	4	b800:0000
05h	graph	320x200	4		b800:0000
06h	graph	640x200	2		b800:0000
07h	text	80x25	mono		b000:0000
0dh	graph	320x200	16	8	a000:0000
0eh	graph	640x200	16	4	a000:0000
0fh	graph	640x350	mono	2	a000:0000
10h	graph	640x350	16	2	a000:0000
11h	graph	640x480	mono		a000:0000
12h	graph	640x480	16		a000:0000
13h	graph	320x200	256		a000:0000

Megjegyzés: A szabványos IBM módok bekapcsolásakor a kép nem törlődik, ha az AL 7. bitjét 1-re állítjuk.

Példa: A 320×200/256-os VGA grafikus mód beállítása:

```
asm
    mov    ax, 0013h
    int    10h
end;
```

SET TEXT-MODE CURSOR SHAPE, a szöveges mód kurzoralakjának beállítása

AH 01h
 CH a 7. bit legyen mindig 0!
 0—4. bit: a kurzor felső, legelső rasztersora a karakteren belül
 5—6. bit: a villogás beállítása: 00 normál, bármely más kombináció esetén nem látható
 CL 0—4. bit: a kurzor alsó, legutolsó rasztersora a karakteren belül

SET CURSOR POSITION, a kurzor pozíciójának beállítása

AH 02h
 BH a lap száma, ahol a kurzort be akarjuk állítani
 0—3 a 2. és a 3. módban
 0—7 a 0. és az 1. módban
 0 grafikus módok esetén
 DH sorszám (Y koordináta)
 DL oszlopszám (X koordináta)

GET CURSOR POSITION AND SIZE, a kurzor pozíciójának és méretének lekérdezése

AH 03h
 BH a lekérdezett lap száma (mint a beállításnál)

Visszaadott értékek:

CH a kurzor legfelső rasztersorának száma
 CL a kurzor legalsó rasztersorának száma
 DH sorszám (Y koordináta)
 DL oszlopszám (X koordináta)

SELECT ACTIVE DISPLAY PAGE, az aktív videolap beállítása

AH 05h
 AL az új aktív lap száma

A funkció hatástalan azokban a videomódokban, ahol csak egyetlen képernyő használható.

SCROLL UP WINDOW, egy ablak felfelé görgetése

AH 06h

AL hány sort gördüljön felfelé az ablak (00 esetén törli a megadott ablakot)

BL az alulról bekerülő üres sor színértéke

CH,CL sor és oszlop (Y,X), az ablak (görgetendő terület) bal felső sarkának koordinátái

DH,DL sor és oszlop (Y,X), az ablak (görgetendő terület) jobb alsó sarkának koordinátái

Megjegyzés: A görgetés csak az aktív lapon végezhető. Néhány implementáció megváltoztathatja a BP regisztert, ezt célszerű ellenőrizni.

SCROLL DOWN WINDOW, egy ablak lefelé görgetése

AH 07h

BL a felülről bekerülő üres sor színértéke

A többi regiszter feltöltése a 06h funkciónál leírtakkal azonos.

READ CHAR AND ATTR AT CURSOR POSITION, a karakter és a színérték leolvasása a kurzor pozíciójában

AH 08h

BH a lekérdezett lap száma (mint a kurzorpozíció beállításakor)

Visszaadott értékek:

AH színbájt

AL karakterkód

WRITE CHAR AND ATTR AT CURSOR POSITION, egy karakter és a színérték kiírása a kurzor pozíciójába

AH 09h

AL a kiírandó karakter kódja

BL színérték, ha grafikus módban a BL 7. bitje 1, akkor a karakter KIZÁRÓ VAGY (XOR) kapcsolatba kerül a képernyővel

BH az írni kívánt lap száma

CX az adott karakter CX-szer íródik ki

WRITE CHAR AT CURSOR POSITION, egy karakter kiírása a kurzor pozíciójába

AH 0ah

AL a kiírandó karakter kódja

BH az írni kívánt lap száma

BL színérték a grafikus módokban (mint a 09-es funkciónál)

CX mint a 09-es funkciónál

SET BACKGROUND/BORDER COLOR, a háttér és a keret színének beállítása

AH 0bh

BH 00h

BL háttér/keret szín (a keret csak szöveges módokban állítható)

SET PALETTE, a paletta beállítása

AH 0bh
BH 01h
BL palettaazonosító
 00h zöld—piros—sárga és háttér
 01h cián—lila—fehér és háttér

Megjegyzés: Ez a funkció a 05-ös grafikus módhoz használatos, még a CGA kártyák idejéből maradt meg, a VGA esetében nem használjuk.

WRITE GRAPHICS PIXEL, egy grafikus képpont (pixel) kigyújtása

AH 0ch
BH az írni kívánt lap száma (természetesen csak azokban a módokban, ahol több grafikus lap is megengedett)
AL a pixel színe [ha a 7. bit 1, akkor a pixel KIZÁRÓ VAGY (XOR) kapcsolatba kerül a képpel, kivéve a 13h módot]
CX a pixel X koordinátája
DX a pixel Y koordinátája

READ GRAPHICS PIXEL, egy grafikus képpont lekérdezése

AH 0dh
BH az olvasni kívánt lap száma
CX a pixel X koordinátája
DX a pixel Y koordinátája

Visszaadott érték:

AL a lekérdezett pixel színe

Megjegyzés: A képpont lekérdezése és kigyújtása funkciók csak grafikus módban működnek.

TELETYPE OUTPUT, „telexszerű” kiírás

AH 0eh
AL a kiírandó karakter kódja
H az írni kívánt lap száma
BL háttérszín grafikus módoknál

Megjegyzés: A funkció abban különbözik a 0ah (karakter kiírása) funkciótól, hogy a szabványos ASCII karaktereket nyomtatja. A 0ah funkció a IBM PC karaktereit írja ki, míg a „telex” kiírás az alábbi karaktereket is felismeri:

07h—BEL, 08h—BS, 0ah—LF, 0dh—CR.

GET CURRENT VIDEO MODE, az aktuális videomód lekérdezése

AH 0fh

Visszaadott értékek:

AH a karakteroszlopok száma

AL az utoljára 00h funkcióval beállított mód kódja

BH az aktív lap száma

SET SINGLE PALETTE REGISTER, egy palettaregiszter beállítása

AX 1000h

BL palettaregiszter/állapotregiszter szám (00h—0fh)

Az elérhető VGA regiszterek (ez a rész nem dokumentált):

10h mód kontroll regiszter

11h keretszín regiszter

12h bitplane-engedélyező regiszter

13h vízszintes pixeles eltolás regiszter

14h színmódkiválasztó regiszter

A palettaregiszterek az EGA kártya maradványai, ott ezek konkrét színeket tartalmaztak, míg a VGA esetén ezek csak mutatók az RGB táblázatra (lásd: színlap kiválasztása, 1013h funkció).

Megjegyzés: Az állapotregiszterekre vonatkozó rész nem dokumentált, ezért ezek átírása ezzel a funkcióval nem javasolt.

SET BORDER COLOR, a keret színének beállítása

AX 1001h

BH keretszín (00h—3fh, 6-bites EGA módszerrel)

SET ALL PALETTE REGISTERS, a teljes paletta beállítása

AX 1002h

ES:DX mutató a palettaadatokra

A palettaadatoknak egy 17 bájtos táblázatban kell lenniük, amelynek az első 16 eleme a palettaszíneket adja, az utolsó bájt pedig a keret színét. A regiszterek értékei mutatók az RGB táblázatra.

TOGGLE INTENSITY/BLINKING BIT, az intenzitás/villogás bit beállítása

AX 1003h

BL 00h = 16 háttérszín villogás nélkül

01h = 8 háttérszín villogással.

A funkció karakteres módban a színbájtok 7. bitjének funkcióját állítja be. Az alapeset mindig a 01h állapot, azaz a villogás+8 háttérszín. A BIOS-ban nincs funkció a legutolsó állapot lekérdezésére, ezt a 0040h:0065h rendszerváltozó 5. bitje jelzi csupán.

GET SINGLE PALETTE REGISTER, egy palettaregiszter lekérdezése

AX 1007h

BL a lekérdezendő palettaregiszter/állapotregiszter száma

Visszaadott érték:

BH a lekérdezett regiszter értéke

A funkció az 1000h funkcióval beállított értékeket tudja lekérdezni, az állapotregiszterek lekérdezése nem dokumentált, ezért ennek eredményében nem lehetünk biztosak.

GET BORDER COLOR, a beállított keretszín lekérdezése

AX 1008h

Visszaadott érték:

BH keretszín

READ ALL PALETTE REGISTERS, a teljes paletta lekérdezése

AX 1009h

ES:DX mutató egy 17 bájtos bufferre

Visszaadott értékek:

Az ES:DX által mutatott 17 bájtos buffer első 16 bájtjában a palettaregiszterek aktuális értékét kapjuk meg, a 17. bájt a keret színét adja.

SET VGA-RGB PALETTE REGISTER, egy VGA-RGB palettaregiszter beállítása

AX 1010h

BX a beállítandó regiszter száma (0..255)

DH a regiszter piros (R) értéke

CH a regiszter zöld (G) értéke

CL a regiszter kék (B) értéke

A funkcióval közvetlenül beállítható egy RGB regiszter, a színt komponensek 6-bitesek, az utolsó két bitet a kártya figyelmen kívül hagyja.

SET BLOCK OF RGB REGISTERS, több RGB palettaregiszter beállítása

AX 1012h

BX a kezdő RGB regiszter száma

CX a beállítandó regiszterek száma

ES:DX mutató egy $3 \cdot CX$ méretű táblázatra, ahol rendre követik egymást a palettaregiszterek R, G és B értékei. A színtkomponensek 6-bitesek.

SELECT COLOR PAGE, színlap kiválasztása

AX 1013h

BL 00h: a lapozás módjának kiválasztása, ekkor:

BH 00h: 4 db 64 bájtos blokk kiválasztása

BH 01h: 16 db 16 bájtos blokk kiválasztása

Ha

BL 01h: egy lap kiválasztása, ekkor:

BH lapszám (0..3 vagy 0..15 a fenti beállítástól függően).

A funkció az egyszerű (EGA) palettaregiszterek értelmezését állítja be. Az első esetben a normál palettaregiszterek 6-bites értékeket tartalmaznak, amelyek a 4 lehetséges RGB lap közül valamelyikre mutatnak. A második esetben maga a palettaregiszter száma a mutató a 16 RGB lap valamelyikének a megfelelő adatára.

READ VGA-RGB REGISTER, egy RGB regiszter lekérdezése

AX 1015h

BX a lekérdezendő RGB regiszter száma (0..255)

Visszaadott értékek:

DH a regiszter piros (R) értéke

CH a regiszter zöld (G) értéke

CL a regiszter kék (B) értéke.

READ BLOCK OF RGB REGISTERS, több RGB regiszter lekérdezése

AX 1017h

BX a legelső lekérdezendő RGB regiszter száma

CX a lekérdezendő regiszterek száma

ES:DX mutató egy $3 \cdot CX$ méretű táblázatra

Visszaadott adatok:

Az ES:DX által megcímezett táblázatba a funkció betölti az RGB regiszterek értékeit R, G és B sorrendben.

GET COLOR PAGE STATE, a színlap beállításának lekérdezése

AX 101Ah

Visszaadott adatok:

BL a lapozás módja:

00h 4 db 64 bájtos lap

01h 16 db 16 bájtos lap

BH az aktuális lap száma

PERFORM GRAY-SCALE SUMMING, szürkeskálává konvertálás

AX 101Bh

BX a legelső konvertálandó regiszter száma

CX a konvertálandó regiszterek száma

A funkció szürkeskálává alakítja az RGB regiszterek színeit.

TEXT MODE CHAR FUNCTIONS, szöveges mód karaktergenerátor funkciók

AH 11h

AL 00h,10h : a felhasználói karakterkészlet betöltése

ES:BP mutató a felhasználói adatokra

CX az átírt kívánt karakterek száma

DX az első átírt kívánt karakter kódja

BL az átírt kívánt karakterblokk száma (0..7)

BH egy karakter magassága : bájt/karakter

AL 01h,11h: ROM monokrom minták betöltése (8x14)

BL a karakterblokk száma

AL 02h,12h: ROM 8x8 -as készlet betöltése

BL a karakterblokk száma

AL 03h: blokkspecifikáció beállítása

BL blokk specifikáció

0, 1, 4. bitek: az aktuális blokk, ha a színbájt 3. bitje 0

2, 3, 5. bitek: az aktuális blokk, ha a színbájt 3. bitje 1

AL 04h: ROM 8x16-os készlet betöltése

BL a karakterblokk száma

GRAPH MODE CHAR FUNCTIONS, grafikus mód karaktergenerátor funkciók

AH 11h

AL 20h: a felhasználói 8x8-as készlet betöltése

ES:BP mutató a felhasználói adatokra

AL 21h: a felhasználói grafikus karakterek betöltése
ES:BP mutató a felhasználói adatokra
CX egy karakter magassága: bájt/karakter
BL sor beállítása
 00h felhasználói beállítás, ekkor a DL a sorok száma
 01h — 14 sor
 02h — 25 sor
 03h — 43 sor

AL 22h: a 8x14-es ROM készlet betöltése
BL sorbeállítás, mint az AL=21h funkciónál

AL 23h: a 8x8-as ROM készlet betöltése
BL sorbeállítás, mint az AL=21h funkciónál

AL 24h: a 8x16-os grafikus ROM készlet betöltése
BL sorbeállítás, mint az AL=21h funkciónál

GET FONT INFORMATION, karakterkészlet információk lekérdezése

AX 1130h
BH mutatóspecifikáció:
 00h az INT 1Fh megszakítás mutatója (pointere)
 01h az INT 43h megszakítás mutatója (pointere)
 02h mutató a ROM 8x14-es készletre (a ROM karakterkészlet címe)
 03h mutató a ROM 8x8-as készletre (a ROM karakterkészlet címe)
 04h mutató a ROM 8x8-as készletre, még hozzá a felső 128 karakterre
 (a ROM karakterkészlet címe)
 05h mutató a ROM 9x14-es készletre (a ROM karakterkészlet címe)
 06h mutató a ROM 8x16-os készletre (a ROM karakterkészlet címe)
 07h mutató a ROM 9x16-os készletre (a ROM karakterkészlet címe)

Visszaadott adatok:

ES:BP a megfelelő mutató
CX az adott készlet bájt/karakter értéke (karaktermagasság)
DL karaktorsorok száma a képen-1

GET EGA/VGA INFO, EGA/VGA alapinformációk

AH 12h
BL 10h

Visszaadott adatok:

BH 00h — a kártya színes módban van
 01h — a kártya monokrom módban van

- BL 00h — a kártyán 64K memória van
- 01h — a kártyán 128K memória van
- 02h — a kártyán 192K memória van
- 03h — a kártyán 256K (vagy több) memória van
- CH feature bitek
- CL kapcsolóállapotok: hardver beállítások

ALTERNATE PRTSC, az alternatív printscreen rutin aktiválása

- AH 12h
- BL 20h

A normál ROM-BIOS rutinok nem biztos, hogy ki tudják nyomtatni a 25 sorosnál nagyobb karakteres képeket. A funkció aktiválja a VGA-BIOS képernyőnyomtató rutinját (INT 5 átirányítása).

SELECT VERTICAL RESOLUTION, a függőleges felbontás kiválasztása*

- AH 12h
- BL 30h
- AL a függőleges felbontás kódja
 - 00h 200 rastersor (CGA minta)
 - 01h 350 rastersor (EGA minta)
 - 02h 400 rastersor

A funkció a karakteres módok számára megadja a függőleges felbontás értékét. A rutin után közvetlenül hívni kell egy szöveges módot beállító 00h funkciót, ekkor lép életbe az új felbontás. Alapesetben a szöveges módok 400 rastersort használnak.

PALETTE LOADING, palettaletöltés engedélyezése/tiltása*

- AH 12h
- BL 31h
- AL 00h engedélyezi az alapbeállítású paletta letöltését
- 01h tiltja az alapbeállítású paletta letöltését

VIDEO ADDRESSING, videocímzés engedélyezése/tiltása*

- AH 12h
- BL 32h
- AL 00h engedélyezi a videocímzést
- 01h tiltja a videocímzést

GRAY-SCALE SUMMING, szürkeskála engedélyezése/tiltása*

- AH 12h
- BL 33h
- AL 00h engedélyezi a szürkeskálát
- 01h tiltja a szürkeskálát

CURSOR EMULATION, kurzoremuláció engedélyezése/tiltása*

AH 12h
BL 34h
AL 00h engedélyezi a kurzoremulációt
01h tiltja a kurzoremulációt

REFRESH CONTROL, frissítés engedélyezése/tiltása*

AH 12h
BL 36h
AL 00h frissítés engedélyezése
01h frissítés tiltása

* A csillaggal jelölt funkciók előbeállítást végeznek, azaz valamely videomód beállítása előtt kell ezeket meghívni, és az utánuk következő 00h módbeállító funkciótól kezdődően fejtik ki hatásukat. Ha például úgy akarunk videomódot váltani, hogy meg akarjuk tartani az előző mód palettáját, akkor a 12h(31h) funkcióval tiltani kell az alapbeállítású paletta betöltődését.

WRITE STRING, szöveg kiírása

AH 13h
AL az írás módja
00h a kurzor pozíciója nem változik meg
01h a kurzor a szöveg végére kerül
BH a lap száma, ahová írni akarunk
BL színbájt, vagyis a megjelenő szöveg színe
CX a sztring hossza
DH,DL sor- és oszloppozíció, ahová a szöveget kiírjuk
ES:BP mutató a kiírandó sztringre (a szöveg szegmens- és offszetcíme).

DISPLAY COMBINATION, képernyőkombináció beállítása/lekérdezése

AH 1ah
AL 00h lekérdezés, ekkor a visszaadott adatok:

BL az aktív kép kódja (lásd lent)
BH az alternatív kép kódja

AL 01h beállítás, ekkor meg kell adni a következő adatokat:
BL az aktív kép kódja (lásd lent)
BH az alternatív kép kódja

Kombinációs kódok:

00h nincs kijelző
01h monokrom adapter/monokrom display

02h	CGA/színes display
03h	fejlesztésre fenntartva
04h	EGA/színes display
05h	EGA/monokrom display
06h	PGA/színes display
07h	VGA/monokrom analóg display
08h	VGA/színes analóg display
09h	fejlesztésre fenntartva
0ah	MCGA/digitális színes display
0bh	MCGA/monokrom analóg display
0ch	MCGA/színes analóg display
FFh	ismeretlen displaytípus

STATE INFORMATION, állapotinformációk

AH 1bh

BX az implementáció típusa

0000h visszatér az állapotinformációkkal

ES:DI mutató egy 64 bájtos táblázatra, ahová az információk kerülnek

Az információs tábla formátuma:

Eltolás	Méret	Információ
00h	DWORD	a statikus funkcionális táblázat címe (lásd lent)
04h	BÁJT	videomód
05h	WORD	az oszlopok száma a képen
07h	WORD	a videobuffer hossza bájtokban
09h	WORD	a videobuffer kezdőcíme
0bh	WORD	a 0. lap kurzorpozíciója
0dh	WORD	az 1. lap kurzorpozíciója
0fh	WORD	a 2. lap kurzorpozíciója
11h	WORD	a 3. lap kurzorpozíciója
13h	WORD	a 4. lap kurzorpozíciója
15h	WORD	a 5. lap kurzorpozíciója
17h	WORD	a 6. lap kurzorpozíciója
19h	WORD	a 7. lap kurzorpozíciója
1bh	WORD	a kurzor típusa
1dh	BÁJT	az aktív videolap száma
1eh	WORD	a CRTC (elektronsugár-vezérlő regiszterek) portcíme
20h	BÁJT	a 3?8 regiszter beállítása
21h	BÁJT	a 3?9 regiszter beállítása
22h	BÁJT	a sorok száma a képen
23h	WORD	bájt/karakter szám

Eltolás	Méret	Információ
25h	BÁJT	displaykombinációs kód az aktív laphoz
26h	BÁJT	az alternatív kép displaykombináció kódja
27h	WORD	az aktív módban használható színek száma
29h	BÁJT	az aktív módban használható lapok száma
2ah	BÁJT	rasztersorszám kód (0,1,2,3) = (200,350,400,480)
2bh	BÁJT	az elsődleges karakterblokk száma
2ch	BÁJT	a másodlagos karakterblokk száma
2dh	BÁJT	vegyes jelzőbitek:
bit	0	minden mód él, minden display él
	1	szürkeskála aktiválva
	2	monokrom display
	3	alapbeállítású paletta letöltése tiltva
	4	kurzoremuláció engedélyezve
	5	villogás mód
	6	PS/2 plazmadisplay aktív
	7	fenntartva
2eh	3 BÁJT	fenntartva (00,00,00)
31h	BÁJT	videomemória 00=64K, 01=128K, 02=192K, 03=256K
32h	BÁJT	az elmentett mutatók állapotbitjei:
bit	0	512 karakter aktív
	1	a dinamikus mentésmező aktív
	2	az alfanumerikus karakterátírás aktív
	3	a grafikus karakterátírás aktív
	4	a palettaátírás aktív
	5	a displaykombináció átírása aktív
	6	fenntartva
	7	fenntartva
33h	13 BÁJT	fenntartva (00...)

A statikus funkcionális táblázat formátuma:

Eltolás	Méret	Információ
00h	BÁJT	a megadott módok installáltak, minden bit egy módot reprezentál
01h	BÁJT	mint a 00h, csak a 08h..0Fh módokat reprezentálja
02h	BÁJT	mint a 00h, csak a 10h..13h módokat reprezentálja
03h	4 BÁJT	fenntartva (00,00,00,00)
07h	BÁJT	változtatható rasztersorszám installálva; bit 0..2=200, 350, 400 rasztersor-beállítás lehetősége
08h	BÁJT	a teljes karakterblokkok száma
09h	BÁJT	az aktiválható karakterblokkok legnagyobb száma
0ah	BÁJT	jelzőbitek #1:

Eltolás	Méret	Információ
bit	0	minden mód és minden display lehetőség adott
	1	a szürkeskála lehetőség adott
	2	a karakterváltoztatás lehetősége adott
	3	a palettaengedélyezés/tiltás lehetősége adott
	4	a kurzoremuláció adott
	5	EGA paletta installálva
	6	színes paletta installálva (RGB)
	7	palettalapozás installálva (4/64 vagy 16/16)
0bh	BÁJT	jelzőbitek #2:
bit	0	fényceruza lehetőség adott
	1	az állapotinformációk mentése/töltése adott
	2	a 16 háttérszín/villogás kapcsolgatása adott
	3	a displaykombináció kódok adottak
	4—7	fenntartva
0ch	BÁJT	fenntartva (00)
0eh	BÁJT	az elmentett mutatók állapotbitjei; ugyanaz, mint az információs táblában
0fh	BÁJT	fenntartva (00)

SAVE/RESTORE VIDEO STATE, a videoinformációk elmentése és visszahívása

AH 1ch

AL 00h

ES:BX mutató egy 64 bájtos táblázatra

Visszaadott értékek:

BX 01h elmentés az ES:BX helyre
02h visszahívás az ES:BX helyről

CX a kért információ típusa

bit 0 videohardver

1 BIOS adatmező

2 a színregiszterek állapota

A BIOS rutinjaira is ugyanaz érvényes, mint a VGA kártya regisztereire: van olyan, amelyet gyakran fogunk használni, és van sok olyan funkció is, amely lényegtelen a mi szempontunkból. A számunkra az a legmegfelelőbb, ha a BIOS rutinok használatát kombináljuk a regiszterek közvetlen programozásával, azaz mindig az éppen célszerűbb módszert alkalmazzuk.

A BIOS rutinokkal (a regiszterekkel ellentétben) bátran lehet kísérletezni, de ne feledjük, hogy ahány kártya, annyi lehetőség. A rutinok és a programok Trident 8900C (1 Mbájt RAM-mal) kártyán futottak, de nem lehet garantálni minden kártyára ugyanezt. Ha valami esetleg nem úgy sikerül, ahogyan azt leírtuk, akkor nem csak a könyv lehet a hibás!

5. A VGA kártya programozása szöveges módban



CHARMODE

5.1. Néhány hasznos szubrutin

A BIOS leírásánál láthattuk, hogy a karakteres képernyőnek alapesetben két fajtája van: 40×25 karakteres kép (0. és 1. mód) és 80×25 karakteres kép (2. és 3. mód). A karakterek mindkét esetben 9×16-os karakterboxban foglalnak helyet. Ez azt jelenti, hogy ezeknek a képeknek a felbontása pixelesen 360×400, illetve 720×400 pixel. A VGA kártya alapesetben mindig a 80×25-ös (720×400) módban indul el. Hogyan lehet akkor ettől eltérő módokat beállítani? (Például a Norton Commanderben is van egy 80×50-es mód, és ennek a BIOS-ban semmi nyoma.) Az itt következő néhány rövid program ezeknek a nem közölt karakteres felbontásoknak a beállítását mutatja be.

Foglaljuk össze, hogy mi az, amit biztosan tudunk:

1. Karakteres módban a VGA kártyának 400 a függőleges felbontása.
2. A vízszintes felbontás 720 és 360 lehet, de a karakterek ekkor 9 pixel szélesek. (Nyolcpixeles karakterek esetében pedig 640 és 320 lehet a vízszintes felbontás.)
3. A kép függőleges felbontása az ismertetett videomódok alapján 200, 350, 400 és 480 lehet.
4. A BIOS karakterkészletet beállító rutinja háromféle karakterkészletet ismer: 8×8-as, 8×14-es és 8×16-os karakterméretű készletet. A 9-pixeles módok is ezeket a készleteket használják.

Ha tehát a függőleges felbontást tetszőlegesen változtatni tudnánk, és ehhez tetszőlegesen lehetne karakterkészletet rendelni, akkor a készlet jó néhány felbontással gazdagodna:

Karakterkészlet	400 rasztersoros kép	480 rasztersoros kép
8×8 (9×8)	80×50 karakter	80×60 karakter
8×14 (9×14)	80×28 karakter	80×34 karakter
8×16 (9×16)	80×25 karakter (alapbeállítás)	80×30 karakter

Nem soroltuk fel a 200 és 350 soros képhez tartozó felbontásokat, de általában nem csökkenteni, hanem növelni szeretnénk a felbontást. Mielőtt rátérnénk a saját programokra, meg kell említenünk a Pascal *Textmode* eljárását, amely hasonlókra képes, csak sajnos nem tudja mindezeket a felbontásokat.

A *Textmode(Font8x8)* 40×50-es képet állít be, míg a *Textmode(CO80 or Font8x8)* a 80×50-es képet eredményezi. Ha a Pascal CRT egységét akarjuk használni, akkor a program eljén mindenképpen ajánlatos kiadni egy *Textmode(CO80 or Font8x8)* utasítást, így az összes CRT egységbeli függvény tud az általunk definiált módokban dolgozni. A 80×60-as módban nem elég a *Textmode(CO80 or Font8x8)* utasítás, mert ebben az esetben az utolsó 10 sor eléréséhez a CRT rutin belső *WindMax* változóját is át kell állítani, mégpedig \$3C3C-re. A *Textmode* eljárással, illetve a *WindMax* változó módosításával becsapjuk a CRT egységet, és a program azt „hiszi”, hogy 80×50-es vagy 80×60-as módban van, akármilyen is a valódi felbontás.

A most következő tíz eljárást a **CHARRES.PAS** programban találhatjuk meg. A program az ENTER lenyomására megmutatja az összes felbontást nyolc- és kilenc-pixeles karakterekkel.

1. A 80×25-ös mód beállítása:

Procedure Char80x25;**Assembler;**

Asm

mov ax,0003h

int 10h {3-as mód: 80x25, 9x16-os box }

End;

A mód beállítása láthatóan nem okoz gondot, tulajdonképpen csak a teljesség kedvéért mutatjuk be. Az eljárás egyszerűen meghívja a BIOS módbeállító rutinját a 03-as szabványos mód beállítására.

2. A 80×28-as mód beállítása:

Procedure Char80x28;**Assembler;**

Asm

mov ax,0003h

int 10h {3-as mód: 80x25, 9x16-os box }

mov ax,1111h { A 8x14-es karaktereket állítjuk be }

xor dx,dx { A legelső beállítandó karakter kódja nulla }

mov bx,0800h { A karakterblokk száma 00, BH érdektelen }

mov cx,256 { 256 darab karaktert kell átírni }

int 10h { A ROM 8x14-es karakterek bekapcsolása }

End;

A program először beállítja a normál 80×25-ös képet, majd a karakterkészletet átdefiniálja 9×16-ról 9×14-re. Mivel a függőleges felbontás nem változott, ezért most INT(400/14), azaz 28 karaktorsor látható a képen.

3. A 80×50-es mód beállítása:

Procedure Char80x50;**Assembler;**

Asm

```

mov ax,0003h
int 10h          { A normál 80x25-ös kép beállítása          }
mov ax,1112h     { Most a 8x8-as karaktereket állítjuk be    }
xor dx,dx
mov bx,0800h
mov cx,256
int 10h          { A ROM 8x8-as karakterek beállítása      }

```

End;

Itt ugyanaz történt, mint az előző esetben: először beállítottuk a 80×25-ös képet, majd a karakterkészletet 8×16-ról átírtuk 8×8-ra, aminek eredményeképpen a függőleges karakteres felbontás 25-ről 50-re nőtt. Mint az előbb, itt sem változtattuk meg a függőleges rasztersorok számát (400/8=50).

A következő módok már 480 rasztersoros módban működnek, ezért itt már szükség van a függőleges felbontás megváltoztatására is. Mivel erre nincs BIOS rutin, ezt magunk fogjuk elvégezni az elektronsugár-vezérlő regiszterek (CRTC) közvetlen átprogramozásával. A 480 soros módhoz tartozó regiszterértékeket a \$12-es videomód segítségével határozhatjuk meg. Beállítjuk ezt a módot (640×480, 16 színű grafikus mód), majd egy egyszerű IN utasítással lekérdezzük az aktuális beállításokat. Ezeket az értékeket persze még kombinálni kell, hogy a szöveges módban is érvényesek legyenek, de ezzel itt most nem kell törődnünk. A regiszterek értékeit — egy memóriatömbbe foglalva — közvetlenül a programok mögött tároljuk (a kódszegmensben), így a rutinokhoz semmiféle változót nem kell deklarálni, önmagukban is működőképeseek.

4. A 80×30-as mód beállítása:

Procedure Char80x30;**Assembler;**

Asm

```

mov ax,0003h
int 10h          { A normál 80x25-ös mód beállítása      }
mov dx,03d4h     { 03D4 = elektronsugár regiszterek      }
mov ax,0e11h     { Engedélyezzük a regiszterek írását     }
out dx,ax        { $11-es regiszter 7. bit 0-ba !!!      }
mov cx,18h       { $18 = 24 db regisztert kell átírni    }
xor al,al
lea si,@dat480   { A tárolt adatok címe a memóriában     }
                 { AL a $3D4 portra, AH a $3D5 portra     }
                 { fog kiíródni !!                        }

```



```

@cikl:
    mov  ah,cs:[ si]      { A következő adat beolvasása      }
    out  dx,ax            { A regiszter átírása                }
    inc  si
    inc  al
    loop @cikl            { Átírja mind a 24 regisztert      }
    call ClockMode480     { Az órajelet meg kell változtatni  }
    jmp  @vege            { Visszatérés                        }
@dat480:
                                {A 480 rasztersoros mód CRTC értékei }
    db   5fh,4fh,50h,82h,55h,81h
    db   0bh,3eh,00h,4fh,0dh,0eh,00h,00h,00h,00h
    db   0eah,8ch,0dfh,28h,1fh,0e7h,04h,0a3h,0ffh
@vege:
    End;

```

A program elején beállítjuk a 80×25-ös módot, majd a 400 rasztersort át kell állítani 480-ra. Először a \$11-es regiszter 7. bitjét kell törölni, hogy hozzáférhessünk a 0—7. regiszterekhez. Ez után egy ciklus következik, amely a regisztereket feltölti a memóriában található értékekkel. Azt már tudjuk, hogy ezeket az értékeket nem szabad bántani, mert a hardver megsérülhet, de mivel mi egy szabványos mód értékeit állítjuk be, ezért itt nem kell tartani meghibásodástól. Az eljárásban található *ClockMode480* szubrutin meghívása a vertikális órajelet kapcsolja át a következőképpen:

Procedure ClockMode480; **Assembler;**

Asm

```

    mov  dx,03c2h  { A misc.output regiszter I/O címe a DX-be      }
    mov  al,0e7h   { A felső két bit a lényeg!!                    }
    out  dx,al     { A 480-as szinkronpolaritások beállítása      }

```

End;

Az eljárással a regiszter értékét állítjuk be a 28 MHz-es 9-pixeles horizontális órajelre (2—3. bit) és a 480 soros szinkronpolaritásokra (6—7. bit). Ha a beállító rutinból ezt kihagyjuk, akkor a képernyő tetején és alján kilóg a beállított kép.

5. A 80×34-es mód beállítása:

Procedure Char80x34; **Assembler;**

Asm

```

    mov  ax,0003h
    int  10h          { Beállítjuk a 80x25-ös módot      }
    mov  ax,1111h     { és a 8x14-es karakterkészletet }
    xor  dx,dx
    mov  bx,0800h

```



```

mov  cx,256
int  10h                { A 8x14-es készlet beállítása      }
...
{ Az előző eljáráshoz hasonlóan a függőleges felbontást átállítjuk
480-ra }
End;

```

A beállítás a 80×30-as beállításhoz hasonlóan zajlott le, kiegészítve a 8×14-es karakterkészlet beállításával.

6. A 80×60-as mód beállítása:

Procedure Char80x60;**Assembler;**

Asm

```

mov  ax,0003h
int  10h                { Alapkép: 80x25                      }
mov  ax,1112h          { A karakterkészletet 8x8-ra váltjuk }
xor  dx,dx
mov  bx,0800h
mov  cx,256
int  10h                { A 8x8-as karakterek beállítása      }
...
{ Az előzőleg már megismert függőleges felbontás átállítása }
End;

```

A beállítás menete megegyezik a 80×34-es mód beállításával.

Az itt felsorolt üzemmódok mindegyikében a vízszintes felbontás 80 karakter, és a karakterek 9 pixel szélesek. A most következő két eljárás a 8- és 9-pixeles szélesség változtatását mutatja be. A váltáshoz két regiszter módosítására van szükség:

- a \$3C2 kimeneti állapotregiszter 2—3. bitjei a horizontális órajelet kapcsolják át a 360/720-as és a 320/640-es felbontások között. Amikor a karakterek 9 pixelesek, akkor a 360/720-as órajelet kell használnunk;
- a \$3C4,\$01 órajel mód sequencer regiszter 0. bitje határozza meg a karakterszélességet.

Tehát ha mindkét regisztert megfelelően módosítjuk, akkor a karakterek szélessége megváltozik. A beállításnál ne felejtsük el, hogy a sequencer órajel mód regiszter néhány VGA kártya esetében igényli a sequencer reset folyamatot (lásd a lemezmellékleten lévő programokban a *SeqRESET* szimbólum szerepét).

7. Átkapcsolás 8 pixel széles karakterekre:

Procedure Box8;**Assembler;** { 8-pixeles karakterekre vált }

Asm

```

mov  dx,$3cc           { Beolvassuk a $3C2 utolsó értékét      }
in   al,dx

```

```

and    al,$f3           { A két érvényes bitet kimaszkolju k      }
mov    dx,$3c2
out    dx,al            { Visszaírjuk az új értéket            }
mov    dx,$3c4          { Sequencer órajel mód regiszter        }
mov    al,1
out    dx,al            { Megcímezzük az órajel mód regisztert   }
inc    dx
in     al,dx            { Beolvassuk az utolsó értékét          }
or     al,1             { A 0. bitet 1-be állítjuk              }
out    dx,al            { Az új érték 8 pixeles módra kapcsol   }
End;

```

Első lépésként beolvassuk a \$3CC regisztert, amely a kimeneti állapotregiszter (\$3C2) olvasási portja. Mivel 8 pixelre akarunk kapcsolni, ezért a 2. és 3. bitbe 0-t kell írni. Ezt logikai ÉS (AND) művelettel végezzük (a két bitet kimaszkoljuk: \$f3=11110011), majd az új értéket kiküldjük a \$3C2 portra. Ezt követően beolvassuk az órajel mód regiszter értékét, és a 0. bitet 1-re állítjuk egy logikai VAGY (OR) művelettel, bekapcsolva ezzel a 8-pixeles karakteres módot.

A visszakapcsolás ezeknek a műveleteknek a megfordításából áll. Először 01-et kell írni a \$3C2 port 2—3. bitjeibe, majd az órajel mód regiszter 0. bitjét 0-ba kell állítani.

8. Átkapcsolás 9 pixel széles karakterekre:

```

Procedure Box9;Assembler;           { 9-pixeles karakterekre vált }
Asm
mov    dx,$3cc          { Beolvassuk a $3C2 utolsó értékét        }
in     al,dx
or     al,4             { Az órajel beállítása a 360/720-as módba }
mov    dx,$3c2
out    dx,al            { Beállítjuk az új értéket a $3C2-be      }
mov    dx,$3c4
mov    al,1             { Az 1-es sequencer regiszter: órajel mód }
out    dx,al
inc    dx
in     al,dx            { Az órajel mód beolvasása                }
and    al,$fe           { Az órajel mód regiszter 0. bitje = 0    }
out    dx,al            { Kivisszük az új értéket                }
End;

```

Ez a két eljárás elvileg korrekt módon elvégzi a 8/9 pixeles váltást, azonban néhány esetben ez még így kevés. A \$3C0, \$13 *horizontális pixeles eltolás* regiszternél ugyanis láthatjuk, hogy a regisztert másképpen kell értelmezni a 8- és a 9-pixeles módban. Ezért ha a karakterek a képernyő szélén esetleg eltolva jelennek meg, akkor ezt a regisztert is módosítani kell az átváltás során.

A vízszintes eltolás inicializálása 8-pixeles mód esetén:

Asm

```

mov    dx,$3da
in     al,dx          { A belső flip-flop törlése          }
mov    dx,$3c0
mov    al,$13         { A $13-as regiszter kiválasztása }
out    dx,al
xor    al,al          { A $13-as regiszterbe 0-t tölt    }
out    dx,al
mov    al,$20         { A képernyő engedélyezése      }
out    dx,al

```

End;

A vízszintes eltolás inicializálása 9-pixeles mód esetén:

Asm

```

mov    dx,$3da
in     al,dx          { A belső flip-flop törlése          }
mov    dx,$3c0
mov    al,$13         { A $13-as regiszter kiválasztása }
out    dx,al
mov    al,8           { A $13-as regiszterbe 8-at tölt    }
out    dx,al
mov    al,$20         { A képernyő engedélyezése      }
out    dx,al

```

End;

Végezetül nézzük meg, hogyan lehet megváltoztatni a színbájt beállítását. A BIOS-nak van egy olyan szolgáltatása, amellyel a színbájtok működését tudjuk megváltoztatni a „8 háttérszín és villogás” és a „16 háttérszín” mód között.

9. A 16 háttér színű mód bekapcsolása:

Procedure Back16;Assembler;

Asm

```

mov    ax,$1003      { A megszakítás funkciókódja az AX-be }
xor    bl,bl         { A BL=00 esetben nincs villogás      }
int    $10           { Beállítjuk a "16 háttérszín" módot  }

```

End;

10. A villogás bekapcsolása:

Procedure Blinking;Assembler;

Asm

```

mov    ax,$1003      { A megszakítás funkciókódja az AX-be }
mov    bl,1          { A BL=01 esetben van villogás      }
int    $10           { Beállítjuk a "villogás" módot      }

```

End;

Ha valaki ragaszkodik ahhoz, hogy ezt regiszteren keresztül végezze el, akkor a megfelelő beállítást a \$3C0, \$10 mód kontroll regiszter 3. bitjének ki-be kapcsolásával végezheti el.

Asm

```

mov  al,$03DA    { A $3DA leolvasása, hogy az attribute cím mód-
                  ban legyen }
in   al,dx
mov  dx,$03C0    { Az attribute regiszterek I/O címe }
mov  al,$10
out  dx,al       { A $10-es regiszter kiválasztása }
inc  dx
in   al,dx       { A $3C1 portról beolvassuk az értékét, }
or   al,8        { vagy "and al,$f7"-ez a bitmódosítás }
dec  dx
out  dx,al       { Kiírjuk a módosított adatot }
mov  al,$20
out  dx,al       { Engedélyezzük a képernyőt (5.bit = 1) }

```

End;



CHARRES.PAS

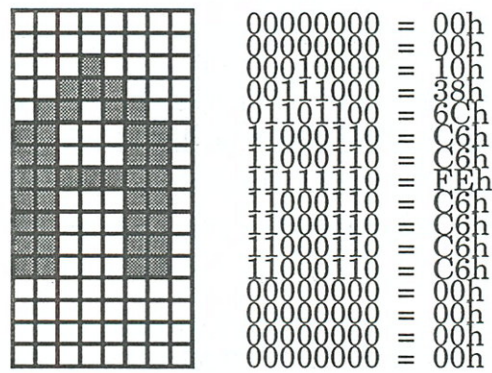
5.2. Felhasználói karakterkészletek

Az előbb, a különböző felbontások beállításánál láthattuk, hogyan lehet aktiválni a VGA kártya ROM-jában lévő beépített karaktereket. A saját készletek definiálására két lehetőség is kínálkozik:

1. A BIOS karakterdefiniáló megszakításával (11h funkció), ugyanúgy, ahogyan a beépített készletek esetében.
2. Közvetlen memóriába írással a karakterkészlet direkt programozásával.

Az egyéni készleteket először persze meg kell tervezni, ehhez pedig ismerni kell a betűket leíró adathalmaz formátumát. A képszerkezetek leírásánál már megismertük a karakterbox fogalmát, azaz az egy karaktert tároló bájtok összefüggő együttesét.

Az "A" betű a memóriában:



17. ábra Karakter a memóriában

A karakter annyi bájtól épül fel, amennyi a függőleges pixeles felbontása. A 17. ábrán egy 8x16-os karakterboxban láthatjuk az „A” betűt. Az összerendelés egyszerű: ahol a karakter tinta színű, ott az adott bit értéke 1, ahol pedig háttér színű, ott a megfelelő bit 0. A bájtok vízszintesen helyezkednek el. Már leírtuk, hogy a 9 pixel széles módok is ezeket a készleteket használják, csupán arra kell ügyelnünk, hogy a készlet \$C0—\$DF elemeinél (ezek a keretrajzoló karakterek) a 9. pixel a 8. pixel értékével egyezni fog. Ha erre nincs szükség, akkor ez ki is kapcsolható (\$3C0, \$10 regiszter 2. bit). A karakterkészlet adathalmaz tehát nem más, mint egy $n \cdot 256$ elemű *bájt* típusú tömb, ahol n a függőleges méret:

Karakterkészlet	Memóriaigény
8×8	256*8=2048 bájt
8×14	256*14=3584 bájt
8×16	256*16=4096 bájt

Nézzük most a karakterdefiniáló eljárást! Legyen ez egy olyan eljárás, amellyel tetszőleges méretű és számú karaktert lehet definiálni.

A paraméterek a következők:

dat a felhasználói adatokra mutató pointer
 car az elsőként átírandó karakter kódja (0..255)
 db az átírandó karakterek száma
 fm a karakter függőleges mérete: bájt/karakter (8,14,16)

Procedure DefChars(dat:pointer; fm:byte; car,db:word);**Assembler;**

Asm

```

push es
push bp          { Tároljuk a szükséges regisztereket      }
mov ax,$1110     { AH=$11,AL=$10: felhasználói definíció  }
mov cx,db        { Ennyi karaktert definiálunk             }
```

```

mov    dx,car          { Ez a kezdő karakter }
xor     bl,bl
mov     bh,fm           { A karakter mérete bájt/karakter-ben }
les     bp,dat          { Az ES:BP mutatót ráállítjuk a készletre }
int     $10             { Átdefiniálás a BIOS rutinnal }
pop     bp
pop     es              { Az elmentett regiszterek visszatöltése }
End;
```

Az eljárással tetszőlegesen átírhatunk egy vagy több karaktert, a függőleges méretet is mi határozzuk meg. Ez azt jelenti, hogy a fenti felbontásokon kívül teljesen egyéni karakteres képet is meg tudunk határozni. Ha pl. 10-re állítjuk a függőleges felbontást (és természetesen az adathalmaz is 10 pixel magas karakterekből áll), akkor a kép mérete 80×40 (400/10) karakter lesz.

A rutin használatára nézzünk egy konkrét példát! Az alábbi program egy négyzetté változtatja a @ jelet:



CHRDEF.PAS

Program Karakter;

Uses Crt;

```

{ Ez az adattömb a karakterboxot tartalmazza; bitminta }
Const negyzet : array[ 0..15] of byte=(
00,$00,$FF,$81,$81,$81,$81,$81,$81,$81,$81,$FF,$00,$00,$00,$00);
```

Var i : integer;

Procedure DefChars(...) { Az előbbi eljárás }

Asm

....

End

BEGIN

Clrscr;

{ Az összes karakter kiírása a képre sárga színnel }

For i:=0 **to** 255 **do** memw[\$b800:i*2]:=\$0e00+i;

Readln;

{ Most átírja a @ karaktert }

DefChars(Addr(negyzet),16,64,1);

Readln;

END.

Ha a Turbo Pascal szövegszerkesztője 25 soros módban van, akkor a program futtatása után az általunk definiált készlet megmarad, azaz a @ karakter helyett egy négyzet fog megjelenni. Ha pl. a hívásban a karakterszámot 1-ről 10-re változtatjuk, akkor a @ utáni karakterek (A, B, C stb.) szintén átíródnak, de mivel a mi tömbünk csak egy karakterboxot tartalmaz, ezért ezek helyén csak valami értelmetlen zagyvaság fog megjelenni.

5.3. A karakterkészlet elérése a memóriában

Az előző program egy BIOS megszakításon keresztül érte el a karakterkészletet, azonban lehetőség van a karaktereknek a memóriában való közvetlen átprogramozására is. A karakterkészlet(ek) a memóriában az A000:0000 címtől kezdődően helyezkednek el. Láthattuk, hogy a karakterek magassága változó. Ez azt eredményezi, hogy a karakterek memóriaigénye is más és más attól függően, hogy milyen magasságot határozzunk meg. A memóriában ezzel szemben minden karakternek egy konkrét 32 bájt terület van fenntartva, azaz bármekkora is a karakter, az egy adott kódhoz tartozó memóriacím $n \cdot 32$, ahol n az adott karakter kódja. Összefoglalva: egy karakter magassága 1 és 32 között szabadon megválasztható (noha az 1 pixel magas karakternek nincs sok értelme), és minden karakterhez egy statikus memóriacím rendelhető. A készletek tárolására 64 Kbájt van fenntartva, ez összesen 2048 ($65536/32=2048$) különböző karakter tárolására elegendő, amiből a kártya egyszerre 512 karaktert tud megjeleníteni. Ebből adódott a BIOS-nál már említett 8 karakterkészlet ($2048/256=8$). A karaktermemóriát azonban nem lehet egyszerű memóriaírással és -olvasással elérni. Már említettük, hogy a bitplane regiszterek minden egyes video-módra hatással vannak. Nem kivétel ez alól a karakteres mód sem. A készletek eléréséhez a következő műveleteket kell elvégeznünk:

1. A bitplane írásmaszkot a 2. bitplane-re kell állítani, mert a karaktereket ezen a bitplane-en tároljuk (\$3C4, \$02 regiszter).
2. A bitplane olvasásmaszkot is erre a bitplane-re kell állítani, de ez természetesen csak akkor szükséges, ha olvasni is akarjuk a karaktermintákat, ami a ritkább eset (\$3CE, \$04 regiszter 4. bit).
3. Tiltani kell a páros/páratlan memóriaelérést, mivel a karakterek bájtfoltyonosak (\$3C4, \$04 regiszter 2. bit).
4. Tiltani kell a páros/páratlan memóriaelérést, mivel a karakterek bájtfoltyonosak (\$3CE, \$05 regiszter 4. bit).
5. A memórialap regisztert az A000 címre kell állítani (\$3CE, \$06).

Bizonyára feltűnik a 3. és a 4. pont azonossága. Ha visszalapozunk a regiszterek leírásához, akkor láthatjuk, hogy a \$3C4, \$04 regiszter 2. bitje és a \$3CE, \$05 regiszter

4. bitje ugyanazt a funkciót látja el. Hogy miért, az a gyártók titka. Vannak kártyák, amelyeken elegendő az egyik regiszter átírása, de van, ahol mindkettő átírására szükség van. Ezt ki lehet kísérletezni, de biztosabb, ha mindkét bitet átállítjuk.

Ha mindezt megtettük, akkor a karakterek elérhetők lesznek az A000:0000 címtől kezdődően, ahol a 0 kódú karakter címe \$0000, az 1 kódú karakteré \$0020 és így tovább. Természetesen a karakter 32 bájtjából csak annyi látszik, amennyi az aktuális magasság. Ha a karakterek módosítását befejeztük, akkor a fenti regiszterek előző értékeit vissza kell állítani, különben a karakteres kép nem lenne elérhető!

Az alábbi program bemutatja mindezt:



CHAR.PAS

```

Program Kar;
uses crt;

var i:word;

Begin
  clrscr;           { Kiírjuk a képre az összes karaktert }
  for i:=0 to 255 do memw[ $b800:i*2] :=$e00+i;

  readln;

  portw[ $3ce] :=$0406; { A memórialapok átírása, az A000:0000 }
                        { cím engedélyezése }
  portw[ $3c4] :=$0402; { A 2. bitplane a karakterek helye }
  portw[ $3c4] :=$0704; { A memória mód átállítása }
  portw[ $3ce] :=$0204; { Olvasás 0. módban a 2. bitplane-ről }
  portw[ $3ce] :=$0005; { A páros/páratlan elérés tiltása(4.bit)}
  for i:=0 to 255 do { A karakterkészlet most elérhető az }
    begin { A000:0000 címtől kezdődően }
      inc(mem[ $a000:0] ); { A 0. karakter legfelső sora }
      dec(mem[ $a000:15] ); { A 0. karakter legalsó sora }
      delay(20);
    end;

  portw[ $3ce] :=$0e06; { A módosított regiszterek visszaírása }
  portw[ $3ce] :=$0004; { az eredeti értékre }
  portw[ $3ce] :=$1005;
  portw[ $3c4] :=$0302;
  portw[ $3c4] :=$0304;
  readln;
End.

```


A program először sárga színnel kiírja a képre a 256 db karaktert. Ez után következik a VGA kártya regisztereinek az átprogramozása a **portw[]** tömb segítségével. Az átprogramozás eredményeképpen a karakterkészlet elérhetővé válik. A belső ciklus a 0 kódú karakter első és utolsó rasztersorát módosítja közvetlen memóriába írással. A program végén a regisztereket visszaállítjuk az eredeti értékekre.

Az előző BIOS megszakításhoz hasonlóan erre a módszerre is írunk egy karakterát-definiáló eljárást.

A paraméterek:

dat a felhasználói karakterre mutató pointer
car az átírandó karakter kódja
fm a karakter függőleges magassága

{ \$G+}

Procedure DefChar(dat:pointer; car, fm:word); **Assembler;**

Asm

```

push ds
push es          { A szegmensregiszterek elmentése          }
mov ax, 0a000h
mov es, ax       { Az ES ráállítása a karakter memóriára    }
lds si, dat      { A DS:SI az új karakteradatra mutat      }
mov di, car      { DI-be a kód                               }
shl di, 5        { 32*kód megadja a karakter memóriacímét   }
                 { ES:DI most a karaktercímet tartalmazza    }
mov cx, fm       { A karakter CX db bájtból áll             }
cld
mov dx, 03ceh
mov ax, 0005h    { A páros/páratlan mód tiltása: 4. bit     }
out dx, ax
mov ax, 0406h    { A memórialap beállítása (A000:0000)      }
out dx, ax
mov dx, 03c4h
mov ax, 0402h    { Bitplane maszk: a 2. bitplane használata }
out dx, ax
mov ax, 0704h    { A páros/páratlan mód tiltása: 2. bit     }
out dx, ax

rep movsb       { A karakteradatok bemásolása a készletbe   }

mov ax, 0302h    { A használt regiszterek visszaírása az    }
out dx, ax       { eredeti értékekre                       }
mov ax, 0304h
out dx, ax
mov dx, 03ceh

```

```

mov  ax,1005h
out  dx,ax
mov  ax,0a06h
out  dx,ax

pop  es      { A szegmensregiszterek visszatöltése }
pop  ds
End;

```

Az eljárás csak egyetlen karakter beállítását végzi el, és még a függőleges karaktermagasságot sem változtatja meg. Első látásra bonyolultabbnak és butábbnak tűnik, mint a BIOS megszakítás. Mindezek ellenére van egy nem elhanyagolható előnye a BIOS rutinnal szemben: a karakter átírásakor nem villan be a kép. Hogy érzékeljük a különbséget, futtassuk le az alábbi két programot:



CHRDEF.PAS

```

1.
for i:=0 to 255 do { Minden karaktert átír }
Begin
  DefChars(addr(negyzet),16,i,1); { A BIOS karaktermódosító funkciója }
  Delay(15);
End;

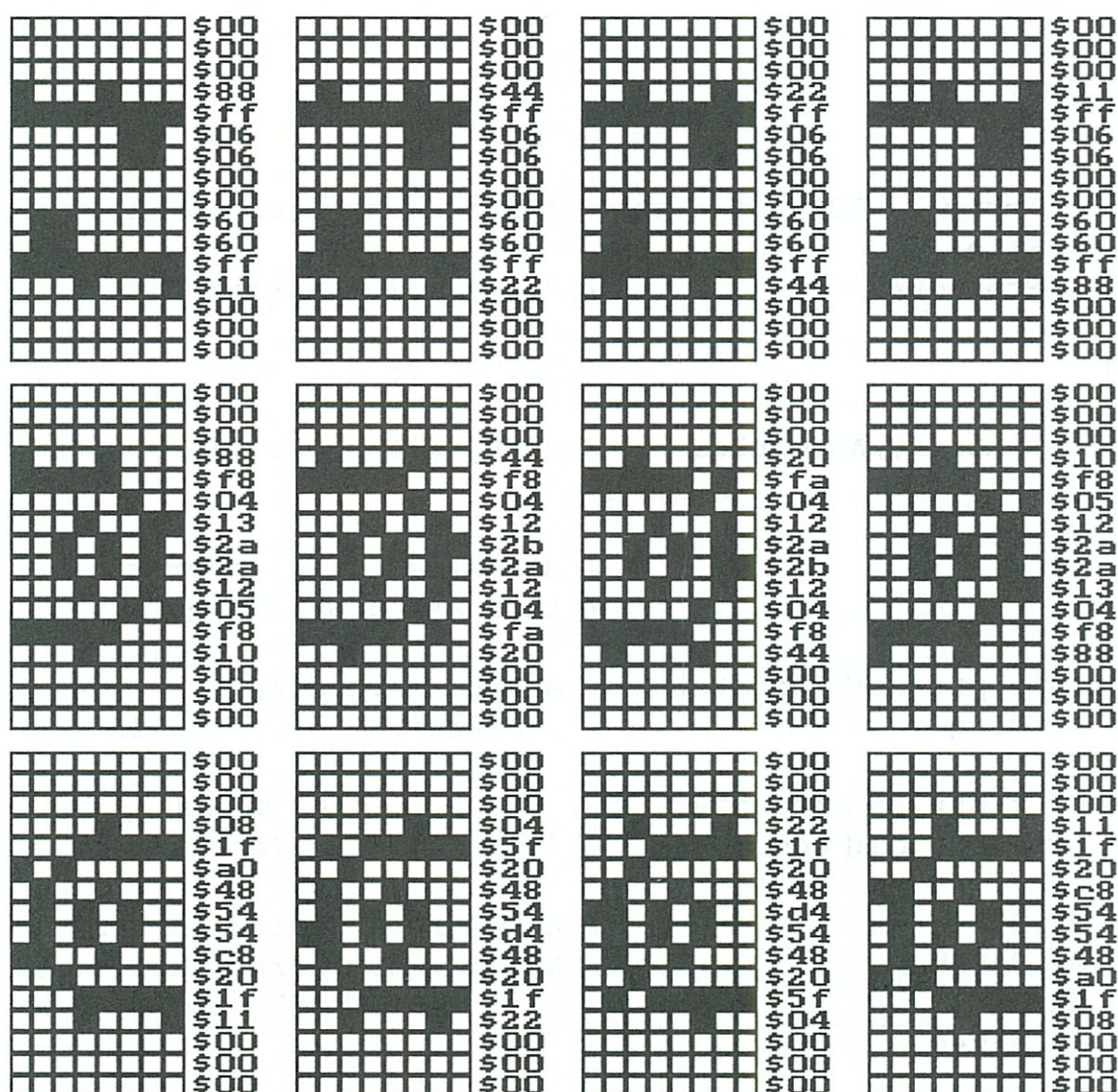
2.
for i:=0 to 255 do { Minden karaktert átír }
Begin
  DefChar(addr(negyzet),i,16);      { A karaktermemória közvetlen
                                   elérése }
  Delay(15);
End;

```

A ciklusokat tegyük be a *Karakter* nevű program definiáló részébe! A képernyőn követhető, ahogyan a rutinok sorban átírják a karaktereket, de az első esetben, amikor a BIOS rutint használjuk, jól látható a zavaró villogás, míg a második esetben ez elmarad. (Figyelem: ha a programokat a fejlesztőrendszerből indítjuk, akkor a futtatás előtt a TP szövegszerkesztőjét kapcsoljuk át 50 soros módba!)

5.4. Karaktermódosítás megszakítás alatt

A most következő példa az első lépés az animáció megismeréséhez. A karakteres képernyő ugyan nem látszik éppen alkalmasnak az animációra, de látni fogjuk, hogy ennek a megoldása sem lehetetlen. Az animáció előre megtervezett mozgásfázisokból épül fel. A következő példaprogram egy futószalag mozgását szimulálja. A futószalagot 3 karakterből építjük fel, és a mozgást 4 fázisra bontjuk.



18. ábra Egy futószalag animációs fázisai

Az első karakter a futószalag egy közbenső elemét ábrázolja, míg a következő két karakter a szalag két végpontját tartalmazza. Minden karakterhez 4 mozgásfázis tartozik, ahol a futószalag pontjai 1-pixeles elmozdulásokat végeznek. Ez a fajta animáció ciklikusan ismétlődik, azaz a negyedik fázis után ismét az első következik, és

így tovább. Az adathalmaz tehát 12 karaktert tartalmaz, minden karakter 16 bájtól épül fel, így a szükséges memória: $12 \cdot 16 = 192$ bájt. A lemez mellékleten ez az adathalmaz *lanc.dat* néven található meg. A mozgáshoz 3 karaktert fogunk felhasználni, a 0. 1. és 2. kódú karaktereket, amelyeknek az átírását közvetlen memóriába írással oldjuk meg. A fázisváltást a *\$1C sorszámú timer* megszakítás fogja elvégezni, amely alapállapotban a felhasználóknak van fenntartva. A megszakítást a számítógép belső órája által ciklikusan kiváltott \$08-as megszakítás aktiválja másodpercenként kb. 18,2-szer, de a megszakítási idő módosítható is (erről majd később lesz szó).



CHRIRQ.PAS

```
{ $G+ }
```

```
Uses Crt, Dos;
```

```
Var i, j : Integer;
    dat : Array[ 0..191] of byte; { A lánckarakterek adatai }
    f : File;
    old : Procedure; { A régi $1C megszakítás }
```

```
Const
```

```
    spd = 1; { Animációs sebesség }
```

```
Procedure TimerIRQ; Assembler; { Az új $1C megszakítás }
```

```
Asm
```

```
    push ds
    push es { A használt regiszterek }
    pusha { tárolása a STACK-ben }
```

```
    dec byte ptr cs:[ @counter] { A sebességszámláló }
    jnz @endirq { csökkentése és vizsgálata }
```

```
    mov byte ptr cs:[ @counter], spd { Inicializálás }
    inc byte ptr cs:[ @counter2] { A fázismutató léptetése }
```

```
    mov dx, 03c4h { A VGA regiszterek átírása, }
    mov ax, 0402h { hogy a karakterkészlet }
    out dx, ax { elérhető legyen }
    mov ax, 0704h
    out dx, ax
    mov dx, 03ceh
    mov ax, 0406h
    out dx, ax
```

```
    mov ax, 0a000h { ES:DI mutat a célcímre, }
    mov es, ax
    mov ax, seg @data
```


[illegible]

```

Procedure Box8;Assembler; { A 8 pixel széles karakterszélesség
                                beállítása }

Asm
    ... Ezt már láthattuk ...
End;

{ Egy futószalag kinyomtatása a képre az X,Y pozícióba
    S színnel }
Procedure Futoszalag(x,y,l,s:byte); { és L hosszúságban }
var b,c:word;
Begin
    c:=x*2+y*80;
    memw[ $b800:c] :=s*256+$02;          { A szalag eleje }
    for b:=1 to l-2 do
        memw[ $b800:c+b*2] :=s*256+$00;    { Közbenső elemek }
        memw[ $b800:c+l*2-2] :=s*256+$01;  { A szalag vége }
End;

BEGIN
    Clrscr;
    Assign(f,'lanc.dat');                  { A futószalag karakter- }
    Reset(f,1);                            { adatait beolvassa a dat }
    Blockread(f,dat,192);                  { nevű tömbbe }
    Close(f);

    Randomize;
    Textmode(c40);                         { 40x25-ös módot használ }

    Getintvec($1c,@old);                   { A régi $1C vektor mentése }
    Setintvec($1c,@TimerIRQ);              { Az új rutin aktiválása }

    Box8;                                  { Átkapcsolás 8 pixel széles
                                            módba }

    { Véletlenszerűen minden sorba kiteszünk egy futószalagot }
    for i:=0 to 24 do
        Futoszalag(1+random(10),i,5+random(25),1+random(15));

    Readln;
    { most a program várakozik az ENTER lenyomására , közben a
      megszakítás végzi az animációt: a szalagok mozognak }

    Setintvec($1c,@old);                   { Az elmentett vektor visszaírása }
END.

```

A program futtatásakor minden sorban megjelenik egy tetszőleges színű és hosszúságú szalag, és minden szalag mozog. Közben a program semmit nem csinál, mert csu-

pán egyetlen READLN utasítás várja, hogy lenyomjuk az ENTER-t. Természetesen ez nem teljesen igaz, mivel a saját \$1C megszakításunk másodpercenként 18-szor végrehajtódik. A program legelején található SPD konstans értékével az animáció sebessége módosítható. A 18 db megszakításból ugyanis csak minden SPD-edik végez fázisváltást, vagyis ha ezt a konstanst megnöveljük, akkor az animáció sebessége csökkenni fog. Maga a fázisváltás egyszerű: a *counter2* nevű fázismutató alsó két bitje megadja az éppen következő fázis számát (0, 1, 2, 3). Ez alapján meghatározzuk a fázisadat címét (ez az SI regiszterbe kerül), majd átmásoljuk a karaktermemóriába az adatokat. A megszakítás egyszerre $3 \cdot 16$, azaz 48 bájt adatot másol át. Ez a módszer egyébként nem a leghatékonyabb megoldás, mert minél nagyobb az animációs karakterek száma, annál több memóriát kell mozgatni minden egyes megszakítás alatt. Másrészt viszont ez a memóriahossz nem függ a fázisok számától.

Egy másik lehetséges megoldás az, hogy (mivel a VGA kártya egyszerre 8 db karakterkészletet tud kezelni) más más készletekben helyezzük el a különböző fázisokat, és nem memóriamásolással oldjuk meg az animációt, hanem a készletek váltogatásával. Ez a megoldás abból a szempontból jobb, hogy memóriamásolás helyett csupán egyetlen regisztert (\$03C4, \$03 Char Map Select) kell módosítani, viszont nyolcra korlátozza a fázisok számát. A másik pedig, amire ilyenkor figyelni kell, hogy azoknak a karaktereknek, amelyek nem végeznek animációt, az összes készletben azonosnak kell lenniük.

5.5. A karakteres képernyő görgetése

Görgetésnek a képernyő egészének vagy valamely részének elmozdítását nevezzük (SCROLL). Mindenki találkozott már vele: amikor egy szövegszerkesztővel dolgozunk, és a szövegben tetszés szerint „mászkalunk”, a kép felfelé vagy lefelé gördül. Általában a karakteres képek esetében a görgetés is karakteres, vagyis az a legkisebb egység, amellyel a kép el tud mozdulni, egy karakter. Az alábbiakban négy olyan egyszerű eljárást ismertetünk, amelyekkel a négy főirányba tudunk mozdtítani egy tetszőleges méretű képterületet (ablakot). A görgetés algoritmusa nem túl bonyolult. A felfelé léptetés során egyszerűen feljebb kell másolni minden sort. Más irányokban is ugyanezt kell tennünk: lefelé léptetésnél minden sor eggyel lejjebb lép, balra léptetésnél az oszlopok balra mozognak, míg jobbra léptetésnél jobbra. A karakteres kép szerkezetét már ismerjük. Ha pl. felfelé kell másolni egy karaktert, akkor az egy WORD típusú adat (szín- és karakterbájt együtt) mozgatását jelenti az *nn* címről az *nn-160* címre (80 oszlopos képernyőkön egy sor 160 bájtból áll). Ezt a sor összes karakterével el kell végezni, hogy az egész sor feljebb lépjen. Majd ezt az egész műveletsort annyiszor kell ismételni, ahány sort feljebb akarunk léptetni. Az ablak leg-

utolsó sorát pedig fel kell tölteni valamilyen színnel és karakterrel, általában szóközzel.

A következő eljárás egy sorral feljebb lépteti az $X1$, $Y1$, $X2$, $Y2$ koordinátákkal megadott ablak karaktereit, a legutolsó sort pedig az E paraméterrel megadott színnel és karakterrel tölti fel. Az $X1$, $Y1$ az ablak bal felső koordinátája, és a Crt unit *WINDOW* eljárásától eltérően a 0,0 a kezdő pozíció. Az $X2$, $Y2$ az ablak jobb alsó sarkának koordinátája, és szintén 0,0 a kezdőértéke. Az egész képernyőt felfelé léptetni tehát a

```
ScrollUp( 0,0 , 79,24 , $0720);
```

utasítással lehet. A \$0720 paraméter \$07-es színt, vagyis világosszürkét és \$20-as karktert, vagyis szóközt jelent.

Hogyan működik az eljárás? Először mindkét adatszegmens regisztert ráállítjuk a képre (az eredeti értékeket tároljuk a veremben), majd kiszámítjuk az ablak bal felső sarkának memóriacímét. A cím a $Cím = 160 * Y + X * 2$ képlettel számítható ki, de ez csak a 80 karakter széles módokban érvényes, mert ekkor található 160 bájt (80 word) egy sorban. Az $Y2 - Y1$ különbség megadja az ablak magasságát, illetve eggyel kevesebbet, de pontosan erre van szükség, hiszen az utolsó sorba már nem mozgatunk másik sort. A magasságot az AX regiszter tárolja. Az ablak szélességét a DX regiszterbe tesszük, ez az $X2 - X1 + 1$ képlettel számítható ki. Ez megadja, hogy hány word típusú adatot kell másolni. A másolást egy ismételt MOVSW utasítással végezzük, majd a címet növelve feljebb léptetünk minden egyes sort. A program végén egy STOSW utasítással az utolsó sort feltöltjük az E értékével.

Procedure ScrollUp($x1, y1, x2, y2, e:word$); **Assembler;**

Asm

```

push  ds      { A Pascal szegmensek tárolása }
push  es
mov    ax,$b800 { Az ES és a DS a }
mov    ds,ax    { karakteres képre mutat }
mov    es,ax
cld
mov    ax,160   { Képcímszámítás }
mul    y1       { AX = 160*Y1 }
mov    bx,x1
shl    bx,1     { BX = X*2 }
add    bx,ax    { BX = 160*Y1+X1*2 }
mov    ax,y2
sub    ax,y1    { AX=Y2-Y1 függőleges méret }
mov    dx,x2
sub    dx,x1
inc    dx       { DX=X2-X1+1 vízszintes méret }
```



```
{ Most következik az ablak soronkénti felfelé másolása }
```

```
@cik:
```

```
mov    cx,dx
mov    di,bx          { ES:DI mutat a célcímre          }
mov    si,bx
add    si,160          { DS:SI mutat a forrásadatokra    }
add    bx,160          { Növeljük a képcímet            }
rep movsw              { Egy sor feljebb másolása        }
dec    ax
jnz    @cik            { AX db sort kell felfelé másolni }
```

```
{ A felfelé másolás után az utolsó sort fel kell tölteni
  az E-vel }
```

```
mov    ax,e            { Karakterkód és színbájt        }
mov    cx,dx
mov    di,bx          { Az utolsó sor feltöltése az      }
rep stosw              { E paraméterrel                  }
pop    es
pop    ds              { A Pascal szegmensek vissza-
                        töltése                          }
```

```
End;
```

A most következő program, amely a *ScrollUp* eljárást is tartalmazza, egy egyszerű demó, amely a görgetés lehetőségeit mutatja be. A programban megtalálható a többi irányba történő görgetés is. A többi irány eljárásai természetesen ugyanazon az elven működnek, mint a felfelé léptetés, csupán az irányok különbözősége miatt adódnak apróbb eltérések a rutinokban, de a módszer mindenütt ugyanaz.

A görgetési lehetőségeket bemutató példaprogram:



SCROLL1.PAS

```
Uses Crt;
```

```
var i,j,k,l,cm:integer;
```

```
const AS:string='Ez itt most egy SCROLL demó !!!      .....'+
                '                                         '+
                'Nem is olyan nagy boszorkányság      .....'+
                '                                         '+
                'Csak ügyesen kell megírni az eljárásokat !'+
                '                                         ';
```

```

Procedure ScrollUp(x1,y1,x2,y2,e:word);Assembler;
  Asm
    { ... megegyezik a fenti eljárással ... }
  End;

Procedure ScrollDn(x1,y1,x2,y2,e:word);Assembler;
  Asm
    ...
  End;

Procedure ScrollLf(x1,y1,x2,y2,e:word);Assembler;
  Asm
    ...
  End;

Procedure ScrollRg(x1,y1,x2,y2,e:word);Assembler;
  Asm
    ...
  End;

Function NextChar:char; { Az AS sztring következő }
  Begin { karakterét veszi ciklikusan }
    Nextchar:=as[ cm] ;
    inc(cm);
    if cm>byte(as[ 0]) then cm:=1;
  End;

BEGIN
  Textmode(300); { 80x50-es módot használunk }

  { Feltöltjük a kép bizonyos sorait különböző színnel és }
  { karakterrel }
  for i:=0 to 4 do
    Begin
      for j:=4 to 75 do
        Begin
          memw[ $b800:i*160+j*2] :=(i+10)*256+254;
          memw[ $b800:(i+20)*160+j*2] :=(14-i)*256+254;

          memw[ $b800:(i+25)*160+j*2] :=(i+1)*256+7;
          memw[ $b800:(i+45)*160+j*2] :=(5-i)*256+7;
        End;
      End;
    End;
  Cm:=1; { Az AS betűmutatót az első karakterre állítjuk }

  Readln; { Várakozunk az ENTER lenyomására }

```



```

{ Most jönnek a SCROLL rutinok }
repeat
  ScrollUp(0,0,4,24,$0f20);
  ScrollRg(0,0,79,4,$0f20);
  ScrollDn(75,0,79,24,$0f20);
  ScrollLf(0,20,79,24,$0f20);

  ScrollDn(0,25,4,49,$0f20);
  ScrollLf(0,25,79,29,$0f20);
  ScrollUp(75,25,79,49,$0f20);
  ScrollRg(0,45,79,49,$0f20);

  { Az AS következő karakterének kiírása a képre }
  inc(1);
  if 1 and 3=0 then
    Begin
      k:=byte(NextChar);
      memw[ $b800:70*2+14*160] :=$f00+k;
      ScrollLf(10,13,71,15,$0720);
      memw[ $b800:44*2+43*160] :=$e00+k;
      ScrollLf(35,34,44,43,$0720);
      ScrollUp(35,34,44,43,$0720);
    End;
  Delay(20);

until keypressed;
{ A demó mindaddig fut, amíg le nem nyomunk egy billentyűt }
{$X+}
  While keypressed do readkey;
{ Kiürítjük a billentyűzetpuffert ; vége }
END.

```

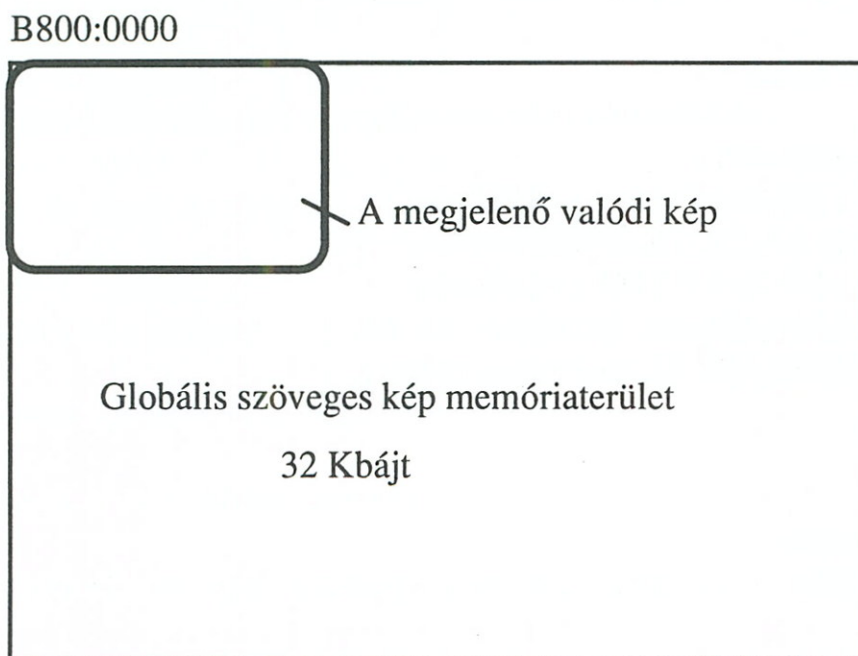
A különböző irányokba való görgetést itt nem ismertetjük, mivel csak apró eltérések vannak, de a **SCROLL1.PAS** program részletesen tartalmazza mindegyiket. A görgető eljárások hasznosak lehetnek más karakteres képet használó programokban is, de a paraméterek értékeit mindig gondosan ellenőrizzük, mivel a program nem vizsgálja a helyességüket. Hibás paraméterek a program lefagyásához vezethetnek (a helyes értékek: $X2 > X1$ és $Y2 > Y1$ és $X=[0..79]$, $Y=[0..24(49)]$).

5.6. Pixeles görgetés karakteres képernyőn

Az előbbi rész rutinjai karakteres elmozdítást eredményeztek, és egy tetszőleges ablakban lehetett használni őket. A most következő program bemutatja, hogyan lehet a karakteres képet pixelesen tologatni. Sajnos ezt csak az egész képernyővel tudjuk

megtenni, illetve osztott képernyős módban a kép felső részével, de erről majd a következő fejezetben lesz szó.

Hogyan lehet elérni a pixeles mozgatót? Tudjuk, hogy a karakteres képernyő a \$B800:0000 címtől kezdődik, és 32 Kbájt tárterületet használhatunk karakteres kép-memóriaként. Ebből a területből a VGA kártya mindig csak egy kisebb részt jelenít meg, méghozzá akkora területet, amennyit az aktuális szöveges mód igényel. Egy 80×25-ös felbontású karakteres kép esetén ez $80 \times 25 \times 2 = 4000$ bájt, míg 40×25-ös képen 2000 bájt. Ez a „részmegjelenítés” úgy is felfogható, mintha lenne egy 32 Kbájtos (globális) karakteres képünk, és ezen, mint egy ablak helyezkedne el a valódi kép.



19. ábra A karakteres videomemória

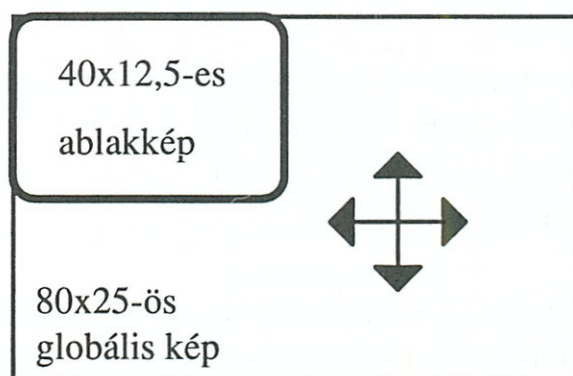
Ha tehát erre a globális képre kiírunk valamit, és az ablakot tetszőlegesen pozicionáljuk rajta, akkor ezt úgy érzékeljük, hogy a képterület az adott irányba mozog. Ezt a pozicionálást kell valahogyan elérni, valamint azt, hogy a képterület globálissá „váljon”. A pozicionálást karakteresen a képkezdőcím regiszterrel (\$3D4, \$0C, \$0D) lehet megoldani, a pixeles pozicionálást pedig két másik regiszterrel kell megadni. A \$3C0, \$13 *horizontális eltolás* regiszter megadja, hogy a kép hány pixellel legyen eltolva balra, tehát ezzel a vízszintes eltolás valósítható meg. Sajnos ez a regiszter *attribute* regiszter, ami azt jelenti, hogy írásakor a képernyő nem látható. Nem éppen kellemes mellékhatás! Szerencsére a VGA kártya meg tudja mondani, hogy az elektronsugár mikor végzi a vízszintes vagy a függőleges visszafutást (ilyenkor nincs képrajzolás). Ha tehát akkor módosítjuk a \$3C0, \$13 regisztert, amikor éppen visszafutás van, akkor ez nem okoz zavart a képernyőn, hiszen az elektronsugár ilyenkor nem rajzol, hanem visszafut. Ez a módszer (mármint a regiszterek visszafutás alatt történő

módosítása) ránk kényszeríti, hogy várakozzunk az elektronsugár visszafutására. A várakozás lassítja ugyan a programot, de nem veszélyes mértékben.

A függőleges pozicionálást a $\$3D4$, $\$08$ Preset Row Scan regiszterrel tudjuk megvalósítani, amelynek alsó 5 bitje megadja, hogy a karakter hányadik rastersorától kezdődjön a kép. Ezeknek a regisztereknek a segítségével pixelesen tudjuk megadni az ablakkép pozícióját a globális képen. Így a görgető eljárás egy megjelenítési rutinra „transzformálódik”, amelynek az X,Y paramétere megadja az ablak pozícióját a globális képen. Legyen az eljárás neve `SHOWPIC(X,Y:Word)`!

Hátravan még a globális kép megadása a VGA kártya számára. Tulajdonképpen a kép alapesetben is globális, csak éppen vízszintes irányban pontosan annyi karakterből áll, amennyi az ablakkép mérete. Ezt kell tehát megnövelni. Az egy sorban elhelyezkedő karakterek számát a $\$3D4$, $\$13$ Offset regiszter adja meg. Ebben a 80 karakteres üzemmód esetén $\$28$, azaz 40 található. Ha visszalapozunk a regiszterek leírásához, akkor láthatjuk, hogy az itt található érték a **bájtok száma egy rastersorban/K**, ahol $K=4$ szócímzés esetén. Így az egy sorban lévő bájtok száma $4 \cdot 40$, azaz 160, tehát az érték helyes.

A következő példaprogram létrehoz egy 80×25 -ös (640×400 pixeles) globális képet, amelyre rámásoljuk a `txtscrn.00` fájl tartalmát. Ez egy 6.0-s Pascal szövegszerkesztő képet tartalmazó fájl. Ezt követően létrehozunk egy 40×12.5 -ös ablakképet, amelynek a pixeles felbontása 320×200 lesz, így az ablak a globális kép egy negyedét mutatja.



20. ábra A pixelscroll megvalósítása

Most bizonyára sokakban felvetődik a kérdés: mi az a 12.5 karakter? Hogy lehet egy képen 12.5 karakter? Nos, lehet. A karakteres kép alapesetben 400 rastersorból épül fel, ezt tudjuk. Ezen a 16 pixeles karakterekből $400/16$, azaz 25 fér el függőlegesen. De ha a rastersorok számát 200-ra tudnánk csökkenteni, akkor azon a 16 pixel magas karakterekből $200/16$, azaz 12.5 férne el. A rastersorokat ugyan nem tudjuk csökkenteni, de a $\$3D4$, $\$09$ -es regiszter (ami egyébként a karakterek magasságát

tartalmazza) 7. bitje bekapcsolja a dupla rasztersoros üzemmódot. Ekkor a 400 sor 200-ra csökken.

Ezek alapján elvégezzük a következő képernyő-beállításokat:

1. Bekapcsoljuk a 80×25-ös módot (\$03), és képre rámásoljuk a *txtscrn.00* fájl tartalmát (a példakép a Pascal szerkesztőből).
2. Átváltunk 40×25-ös módba, de úgy, hogy közben nem töröljük le a képernyőt: az AL 7. bitje 1 (lásd BIOS leírás)!
3. Mivel a 40×25-ös kép felülírja az *Offset* (\$3D4, \$13) regisztert, ebbe visszatöltjük a \$28 értéket, hogy a globális kép 80 karakter széles legyen.
4. A \$3D4, \$09 regiszter 7. bitjét 1-be állítjuk, ezzel 200 soros módba kapcsoljuk a képet.

A négy pont végrehajtása után lesz egy 40×12.5 karakteres ablakképünk és egy 80×25-ös globális képünk. A képernyőn a globális kép egy negyede látszik.

Térjünk vissza a *ShowPic* eljáráshoz! Az eljárásban a következő lépéseket kell elvégezni:

1. Az *X,Y* paraméter alapján meg kell határozni a képcímet, ez kerül a \$3D4, \$0C, \$0D címregiszterekbe. A képen a karakterek 8×16 pixelesek, tehát az *X* alsó 3 bitje a karakteren belüli *X* irányú pixeles eltolást tárolja, míg az *Y* alsó 4 bitje a karakteren belüli *Y* irányú pixeles eltolást. A karakter címét a $Cím = 80 * (Y \text{ shr } 4) + (X \text{ shr } 3)$ képlettel számíthatjuk ki.
2. Az *X,Y* paraméter alapján meg kell határozni a karakteren belüli pixeles eltolásokat: $Xp = X \text{ and } 7$; $Yp = Y \text{ and } 15$
3. A kiszámított értékeket a megfelelő regiszterekbe kell másolni:
Cím: \$3D4, \$0C, \$0D (Start Address High, Low)
Xp: \$3C0, \$13 (Horizontal Pixel Scroll)
Yp: \$3D4, \$08 (Preset Row Scan)

Az eljárás mindig elmenti ezeknek az adatoknak az utolsó értékeit, mivel nem biztos, hogy minden adat megváltozik két hívás között. Ezért az eljárás csak akkor írja újra a regisztereket, ha az értékük a számítás szerint megváltozott. Az eljárás a \$3DA státuszregiszter 3. bitjének tesztelésével érzékeli a vertikális visszafutást.

Procedure ShowPic(*x,y:word*); **Assembler;**

Asm

```

MOV    CX,X           { CX = X koordináta }
MOV    BH,CL           { BH and 7: az X pixeles eltolás 0..7 }
MOV    AX,Y           { AX = Y koordináta }
MOV    BL,AL           { BL and 15: az Y pixeles eltolás 0..15 }
SHR    AX,4            { A pixeles eltolást eltávolítjuk }
```



```

MOV    DX,80          { Egy sorban 80 word helyezkedik el }
MUL    DX              { AX = 80 *(Y shr 4) }
SHR    CX,3           { A pixeles eltolást eltávolítjuk }
ADD    CX,AX           { CX = 80*(Y shr 4) + ( X hr 3) }
                        { Most a CX-ben van a karakter címe }
AND    BX,$070f        { Xp és Yp a BH,BL regiszterekben }
{ A kiszámított címet összehasonlítjuk az utolsó értékkel }
CMP    CX,LAD          { Egyezik-e az utolsó beállított
                        értékkel ? }
JZ     @L1             { Ugrás tovább, ha egyezik }
{ Csak akkor írja át a címregisztereket, ha változás volt }
MOV    LAD,CX          { Tároljuk az új képcímet a LAD
                        változóban }
MOV    AL,$0C          { Az address HI regiszter száma
                        az AL-be }
MOV    AH,CH           { Az AH-ba a cím felső bájtja }
MOV    DX,$03D4
OUT    DX,AX           { Az ADDRESS HI kivitele a regiszterbe }
INC    AL              { Az address LO regiszter száma
                        az AL-be }
MOV    AH,CL           { Az AH-ba a cím alsó bájtja }
OUT    DX,AX           { Az ADDRESS LO kivitele a regiszterbe }
@L1:
CMP    BL,YS           { Az Yp-t is összehasonlítja }
JZ     @L2             { Ugrás, ha egyezik az utoljára
                        beállítottal }
MOV    YS,BL           { Különben tároljuk az új értéket }

MOV    DX,$03DA        { Várunk egy vertikális visszafutásra }
@WAIT1:
IN     AL,DX           { A $3DA 3. bitje 1, ha visszafutás van }
AND    AL,8
JZ     @WAIT1          { Visszaugrás, ha nincs visszafutás }
MOV    DX,$03D4        { Az Y pixeles eltolás a $3D4, $08
                        regiszterbe }

MOV    AL,$08
MOV    AH,BL
OUT    DX,AX           { Az érték kivitele }
@L2:
CMP    BH,XS           { Az Xp-t összehasonlítja az utolsó
                        értékkel }
JZ     @L3             { Ugrás tovább, ha a kettő egyezik }
MOV    XS,BH           { Különben az új értéket tárolja }

MOV    DX,$03DA        { Várakozás a vertikális visszafutásra }

```

```

@WAIT2:
    IN      AL,DX
    AND     AL,8
    JZ      @WAIT2
    MOV     DX,$03C0      { Az attribute regiszterek I/O portja
                           $3C0 }
    MOV     AL,$13        { A horizontális eltolás regiszter
                           a $13-as }
    OUT     DX,AL          { A regiszter kiválasztása }
    MOV     AL,BH
    OUT     DX,AL          { Az Xp kivitele a regiszterbe }
    MOV     AL,$20        { A kép engedélyezése (5. bit=1) }
    OUT     DX,AL

@L3:
    End;

```

Az eljárás az *XS* és *YS* (bájt) változókat használja a pixeles eltolás tárolására, valamint a *LAD* (word) változót az utolsó képcím tárolására. Nézzük meg a teljes programot:



SCROLL2.PAS

```

{ $G+}
Uses Crt,Dos;

var   f   : file;
        i   : integer;
        lad : word;           { Képcímtároló a ShowPic rutinhoz }
        xs,ys : byte;         { Pixeles eltolások tárolása(ShowPic)}
        x,y   : word;         { Ablakkép koordináták }
        Ktb   : array[ 0..127] of boolean; { Billentyűzettáblázat }
        oldKey : procedure;

{ Saját billentyűzetmegszakítás rutin. A megszakítás a KTB }
{ tömb elemeit állítja a SCAN kódok szerint }
Procedure KeyIRQ; Assembler;
Asm
    push    DS                { A használt regiszterek elmentése }
    push    ax
    push    bx
    push    cx
    mov     ax,SEG Ktb        { Inicializálások: }
    mov     DS,ax             { A DS-t az adatszegmensre állítjuk }
    xor     bh,bh             { A BH és a CL regiszterek nullázása }

```



```

mov     cl,bh
in      al,60H      { A billentyű SCAN kód beolvasása }
mov     bl,al       { A BL-be is betesszük a kódot }
and     bl,7fH      { A 7. bitet töröljük }
shl     al,1        { A (lenyomás/felengedés) a CY-be }
cmc     { Komplementáljuk a CY-t }
adc     cl,00        { Tárolás a táblázatban }
mov     [bx+OFFSET Ktb],cl { A táblázat megfelelő elemét állítja }
in      al,61H      { A megszakítási csatorna }
mov     ah,al       { visszaállítása: IRQ vége }
or      al,80H
out     61H,al
mov     al,ah
nop                    { Rövid várakozás }
nop
nop
out     61H,al       { Billentyűzetcsatorna-nyugtázás }
cli
mov     al,20H      { IRQ-vezérlő "IRQEND" üzenet }
out     20H,al
sti
pop     cx          { Regiszterek vissza a veremből }
pop     bx
pop     ax
pop     DS
iret              { A megszakítás vége }
End;

```

Procedure ShowPic(x,y:word); **Assembler;**

Asm

...

End; { Ez a fentebb ismertetett eljárás }

Procedure Box8; **Assembler;** { A 8-pixeles karakterszélesség
beállítása }

Asm

```

mov     al,063h      { A horizontális órajel 25 MHz,
                     (320/640) }

```

```

mov     dx,03C2h

```

```

out     dx,al

```

```

mov     dx,03C4h      { Az órajel mód regiszter 0. bitje = 1 }

```

```

mov     ax,0901h

```

```

out     dx,ax

```

End;

```

BEGIN (*****)
  Clrscr;
  Assign(f,'txtscrn.00');Reset(f,1);
  Blockread(f,mem[$b800:0],4000);
  Close(f); { A demokép beolvasása lemezzről }
  gotoxy(1,1);Readln; { Várakozás az ENTER-re }
  Box8; { A 8-pixeles mód bekapcsolása }
Asm
  mov ax,$0080 { A 40x25-ös mód beállítása a kép }
  int 10h { törlése nélkül ( AL 7. bit 1 ) }
  mov dx,$3d4 { Az offset regiszter visszaállítása }
  mov ax,$2813 { a 80 soros mód értékére }
  out dx,ax
  mov ax,$cf09 { A 200 rasztersoros mód beállítása }
  out dx,ax
End;
  x:=0;y:=0; { Az ablakkoordináta alaphelyzete }
  Showpic(x,y); { A kép megjelenítése }
  FillChar(Ktb,128,0); { A billentyűzettáblázat törlése }
  GetintVec($09,@oldKEY); { A régi $09-es vektor tárolása }
  SetintVec($09,@KeyIRQ); { A saját megszakítás aktiválása }
repeat { A kurzorbillentyűk figyelése }
{ Most a kurzorral SCROLL-ozhatunk a globális képen }
  if Ktb[ 72] and (y>0) then dec(y,2); { fel }
  if Ktb[ 77] and (x<320) then inc(x,2); { jobbra }
  if Ktb[ 75] and (x>0) then dec(x,2); { le }
  if Ktb[ 80] and (y<200) then inc(y,2); { balra }
  Showpic(x,y);
  Delay(1);
until Ktb[ 1]; { Az ESC billentyűre fog kilépni }
  SetintVec($09,@oldKEY); { A régi $09-es vektor visszaírása }
Asm { Ismét 80x25-ös módba kapcsol }
  mov ax,$0083
  int 10h
End;
END. (*****)

```

A program indításakor megjelenik a lemezzről betöltött kép 80×25-ös módban. Nyomjuk le az ENTER-t! A kép átkapcsol 40×12.5-es módba, így az eredeti kép bal felső negyedét látjuk. A kurzorbillentyűkkel a kép mozgatható, két billentyű lenyomásakor a görgetés kétirányú (45 fokos). A programból az ESC billentyűvel lehet kilépni. Előfordulhat, hogy a futás után a billentyűzet nem reagál a gombnyomásokra, ekkor egy CTRL—F10 kombinációt nyomjunk, erre visszaáll az eredeti állapot. Ennek az az oka, hogy a programban saját megszakítással oldottuk meg a billentyűzet lekérdezését, amelynek a működése talán magyarázatra szorul. Tegyük most egy kis kitérőt, és ismerkedjünk meg a billentyűzet működésével és programozásával!

5.7. A billentyűzetmegszakítás működése, saját billentyűkezelés

Mi történik akkor, amikor lenyomunk egy billentyűt? A billentyűzet intelligens bemeneti egység, amely megszakításos adatkapcsolatot tart fenn a processzorral. Amikor megnyomunk vagy felengedünk egy billentyűt, akkor a billentyűzet egy IRQ vonalon megszakítást kér. A megszakításvezérlő áramkör a \$09-es megszakítási vektort rendeli hozzá a billentyűzethez, azaz minden lenyomáskor és felengedésakor egy INT \$09 generálódik. A ROM-BIOS beépített rutinja beolvassa az információt, előállítja a karakter ASCII kódját, és tárolja a karaktert a pufferben. A rutin megfelel gépelésre vagy általános beviteli funkciókra, de nem képes több billentyű egyidejű lenyomását érzékelni. A mi saját rutinunk másként dolgozik. A billentyűzet minden egyes gombjához hozzárendelünk egy logikai változót, amelynek az értéke akkor lesz igaz, ha a billentyűt lenyomtuk. A táblázat nem az ASCII kódok, hanem az ún. *SCAN* kódok szerint tartalmazza a billentyűket.

Maga a megszakítás a következőképpen működik. A megszakítás a billentyűzet adatportról (portcíme \$60) beolvassa az érkező adatot. A bájts alsó 7 bitje a *SCAN* kódot tartalmazza, míg a 7. bit a lenyomás/felengedés információt hordozza. A *SCAN* kód közvetlenül megadja, hogy a *KTB* táblázat melyik elemét kell állítani, a 7. bit pedig azt mondja meg, hogy igazra vagy hamisra kell-e változtatni ezt az elemet. Így a billentyűzet összes eleme egyszerre figyelhető. Ha pl. a *KTB*[72] és a *KTB*[77] egyszerre igaz, akkor ez azt jelenti, hogy a FEL és a JOBBRA billentyűk egyaránt lenyomott állapotban vannak. A *KTB* táblázat módosítása után a megszakítás még alapállapotba hozza a megszakítási csatornát, s ezzel jelzi a megszakításvezérlő áramkörnek, hogy a megszakítás véget ért.

A *SCAN* kódok és a billentyűk összerendelésére nézzünk egy rövid példaprogramot, amely egy adott billentyű *SCAN* kódját adja meg. A program indulása után megjelennek a *SCAN* kódok, hexadecimálisan. Minden billentyű lenyomásakor valamelyik kód után megjelenik egy négyzet, ez jelzi, hogy az adott billentyűnek ez a kódja. Érdeemes megfigyelni, hogy a NumLock állásától függően a vezérlőbillentyűk (kurzor, PgUp, PgDn stb.) lenyomásakor a bal oldali Shift gomb (\$2A) is lenyomást jelez, holott ezt a billentyűt nem is nyomtuk meg.



SCANCODE.PAS

{ \$G+}

Uses Crt,Dos;**var**

i : integer;

Ktb : array[0..127] of boolean;

oldKey : procedure;

const

hh : array[0..15] of char='0123456789abcdef';

Function Hex(x:byte):String; { DEC->HEX konvertálás }**Begin**

hex:=hh[x shr 4] +hh[x and 15] ;

End;

{ Ez a rutin megegyezik az előbbi SCROLL programban található }

{ billentyűzetmegszakítás rutinnal }

Procedure KeyIRQ; **Assembler;****Asm**

....

End;

(*****)

BEGIN

Clrscr;

GetintVec(\$09,@oldKEY); { A \$09 vektor tárolása }

SetintVec(\$09,@KeyIRQ); { A saját rutin aktiválása }

{ Kiírjuk a SCAN kódokat }

For i:=1 **to** 127 **do**

write('\$'+hex(i),': '); {4 db SPACE van a szövegben !! }

Writeln;

Writeln;

Writeln;

Textattr:=\$F;

Writeln(' Kiépés: CTRL-ESC ');

{ Ez a ciklus végzi a billentyűk lekérdezését }

{ Minden lenyomott billentyűt egy / karakterrel jelez }

Repeat**For** i:=1 **to** 127 **do****if** ktb[i] **then** memw[\$b800:(i-1)*16+10] :=\$0edb**else** memw[\$b800:(i-1)*16+10] :=\$0000;**Until** KTB[1] **and** KTB[\$1D] ;

{ Kilépés a CTRL-ESC kombinációra }

{ A \$09 vektort visszaírja az eredeti értékre }

SetintVec(\$09,@oldKEY);

END.

(*****)

A megszakítások vezérlésének cseréje valamilyen oknál fogva nem teljesen korrekt, ezért előfordulhat, hogy a normál billentyűzetrutinra való visszatéréskor a billentyűk nem érzékelik a lenyomást, illetve hibásan működnek. Szerencsére ezt ki lehet küszöbölni úgy, hogy a visszatérés valamilyen CTRL billentyűkombinációra történjen, ilyenkor a vezérlés rendesen visszaáll. Bizonyos áramköri megoldások miatt néhány billentyű kombinálása nem lehetséges. Ha pl. egyszerre megnyomjuk a Z,X,C billentyűket, akkor ezt még pontosan érzékeli a program, de ha a Z,X,C,V,B,N billentyűket nyomjuk le egyszerre, akkor már a SPACE lenyomását nem érzékeli a gép. Sajnos ez hardverprobléma, de általában nem veszélyes, mivel csak három-négy billentyű egyszeri lenyomásakor jelentkezik. Ezt a fajta billentyűzetkezelést a játékprogramokban alkalmazzák előszeretettel, hiszen ott gyakran van szükség több billentyű egyidejű figyelésére. Adódhat olyan eset is, hogy a normál és a módosított rutint egyszerre akarjuk használni. Ezt úgy célszerű megoldani, hogy a saját kezelő eljárást hajtadjuk végre elsőnek. Ekkor a billentyűzetcsatorna és a megszakításvezérlő beállításait nem kell elvégezni, és a regiszterek visszahívása után nem IRET utasítást kell kiadni, hanem meg kell hívni az eredeti BIOS rutint is. Ez úgy a legegyszerűbb, ha valamelyik szabad megszakításvektort ráállítjuk az eredeti rutinra:

```
Getintvec($09,@OldKeyIRQ); { Lekérdezzük a BIOS rutin címét }
Setintvec($60,@OldKeyIRQ); { Ráállítjuk az INT 60h vektort }
Setintvec($09,$NewKeyIRQ); { Az új rutinra pedig az INT 09h-t }
```

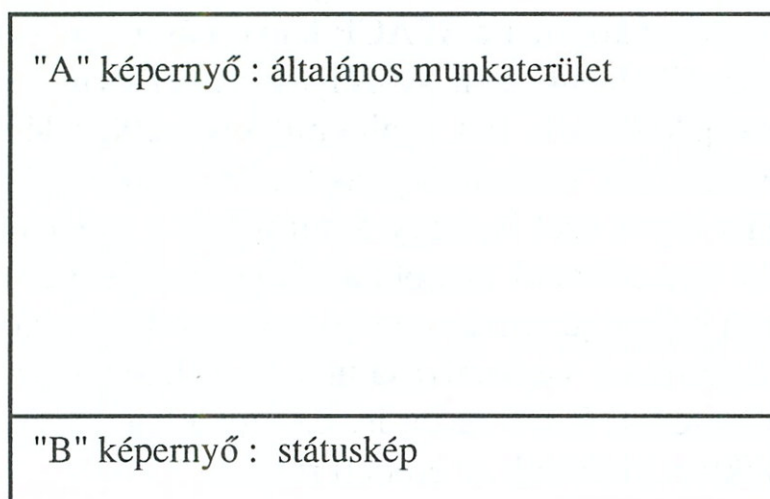
Az új rutinban pedig a következő módosításra van szükség:

```
Procedure NewKeyIRQ; Assembler;
Asm
    push regiszterek
    ...          { Saját megszakítás, az IRQ vonalak nyugtázása
                  nélkül }
    pop regiszterek
    int 60h      { A BIOS eredeti megszakításának hívása }
    iret
End;
```

Így összeláncoltuk a két megszakítást, és lehetőség van az eredeti megszakítás használatára, ugyanakkor a több billentyű egyszerre való figyelése is megoldott.

5.8. Osztott képernyő karakteres módban

A pixeles eltolás bemutatásánál már utaltunk az osztott képernyőre, mint a pixeles megjelenítés másik lehetőségére. Az osztott képernyős módban a VGA kártya két egymástól független képet jelenít meg a monitoron.



21. ábra Az osztott képernyős üzemmód

A két kép együttesen betölti a teljes képernyőt, azaz magasságuk alapesetben 25 karakter. Az „A” kép memóriabeli kezdőcímét a \$3D4, \$0C, \$0D címregiszterek adják meg, tehát ez tetszőlegesen beállítható. A „B” kép mindig a \$B800:\$0000 címen kezdődik. A „B” kép kötöttsége miatt általában csak státusképnek, információk megjelenítésére alkalmas, míg az „A” kép ugyanúgy programozható, mintha a VGA normál karakteres üzemmódban lenne. Tehát az „A” képen megvalósítható a pixeles görgetés, míg a „B” képre mindenféle kiírásokat tudunk elhelyezni. Az osztott képernyős üzemmód tulajdonképpen kettévágja a képet (az angol elnevezése *splitscreen*). Hogy ez a vágás hol történjen a képen, azaz hol érjen véget az „A” kép, és hol kezdődjön el a „B” kép, azt a \$3D4, \$18 *Line Compare* regiszterrel adhatjuk meg. A VGA kártya igen egyszerűen kezeli ezt a regisztert: amikor a rasztensor számláló eléri az itt található értéket, akkor nullázza a címregisztert, így onnantól a „B” kép megjelenítése kezdődik el. Természetesen a vertikális visszafutás alkalmával ez a regiszter visszakapja az eredeti értékét, így mi ebből semmit nem veszünk észre. A regiszter 10-bites, azaz 0 és 1023 közötti értékek tárolására alkalmas. A 8. bit a \$3D4, \$07 *Overflow* regiszter 4. bitjében, míg a 9. bit a \$3D4, \$09 *Maximum Scan Line* regiszter 6. bitjében található. Alapesetben a regiszter minden bitje 1, azaz nem következik be a váltás, mivel a rasztensor számláló sohasem éri el az 1023-as értéket.

Az osztott kép használatát bemutató program ugyanazt teszi, mint a pixeles görgetés példaprogramja. Megjelenítjük a *txtscrn.00* képet, létrehozunk egy globális képet,

majd átkapcsolunk 40×12.5-ös módba. Átírjuk a *Line Compare* regisztert, ezzel már be is kapcsoltuk az osztott képernyős módot. A felső kép pixelesen tologatható a képen, míg az alsó kép egy helyben marad.

A teljes program tulajdonképpen az előző SCROLL2.PAS módosított változata, ezért csak az eltérő részeket ismertetjük.

```
{ $G+}
Uses Crt,Dos;

var   f   : file;
        i   : integer;
        lad : word;
        xs,ys : byte;
        x,y   : word;
        c     : char;
        Ktb   : array[ 0..127] of boolean;
        oldKey : procedure;

Procedure KeyIRQ; Assembler;      { Az előzőleg már ismertetett      }
Asm                                     { billentyűzetmegszakítás      }
    ....
End;

Procedure ShowPic(x,y:word); Assembler; { Az "A" kép pixeles          }
                                           { megjelenítése              }

    Asm
    End;
```

A *ShowPic* eljárást a „B” kép miatt egy kissé módosítani kell, ugyanis a „B” kép, mint az előzőekben láthattuk, mindig a képmemória \$0000 offsetjén helyezkedik el, a címzés szempontjából az „A” kép előtt. Ezt úgy is fel lehet fogni, hogy az „A” kép annyi-
val lejjebb van, amilyen magas a „B” kép. Az „A” képet ezúttal úgy helyeztük el, hogy 80 pixellel lejjebb legyen a „B” képnél (ez öt karaktert jelent), ezért a *ShowPic* elején az *Y* koordinátát növelnünk kell 80-nal. Így végső soron ez a *ShowPic* egy **ADD AX,80** utasításban tér el a **SCROLL2.PAS**-ban találhatóától.



SCROLL3.PAS

```
Procedure Box8;Assembler;
Asm
    ....
End;
```

BEGIN

```

Clrscr;
Assign(f, 'txtscrn.00');
Reset(f, 1);
Blockread(f, mem[ $b800:5*160] , 4000);
Close(f);
gotoxy(1,1);
Readkey;
Box8;

```

```

{ A program elején a "B" képre kiírunk egy szöveget }
TextAttr:=$4f;
Writeln('=====');
Writeln('=          Ez itt a B kép          =');
Writeln('=====');

```

```

{ Elvégezzük a szükséges inicializálásokat a regiszterekben }

```

Asm

```

mov ax,$0080
int 10h      { Átváltás a 40x25-ös text módba
              (képtörlés nélkül) }

mov dx,$3d4
mov ax,$2813 { Az offset regisztert visszaállítjuk }
out dx,ax
mov ax,$cf09 { A rastersorokat megduplázzuk (200) }
out dx,ax
mov dx,$3da
in al,dx     { Az attribute regiszterek cím módba
              kapcsolása }

mov dx,$3c0
mov al,$10   { Mód kontroll regiszter }
out dx,al
inc dx
in al,dx     { Beolvassuk az értékét }
dec dx
or al,32     { Az alsó kép SCROLL-ját letiltjuk (5.bit=1) }
out dx,al    { Kivisszük az új értéket }
xor al,al    { A 0. palettaregiszter kiválasztása }
out dx,al
mov al,56    { Szürke színt viszünk ki, ez a háttér színe }
              lesz }

out dx,al
mov al,$20   { A képet láthatóvá tesszük (5.bit=1) }
out dx,al

```

End;


```

{ Az osztott képernyő installálása: a Line Compare regiszter      }
{ beállítása $0138-ra, azaz 312-re                                }
portw[ $3d4 ] := $1007;      { OVERFLOW 4.bit=1                8. bit }
portw[ $3d4 ] := $3818;      { LINE COMPARE                    0-7. bit }
portw[ $3d4 ] := $8f09;      { MAXIMUM SCAN LINE 6.bit=0      9. bit }

```

```

x:=0;y:=0;
Showpic(x,y);

```

```

FillChar(Ktb,128,0);
GetintVec($09,@oldKEY);
SetintVec($09,@KeyIRQ);

```

```

repeat
  if Ktb[ 72] and (y>0) then dec(y,2);
  if Ktb[ 77] and (x<320) then inc(x,2);
  if Ktb[ 75] and (x>0) then dec(x,2);
  if Ktb[ 80] and (y<256) then inc(y,2);
  Showpic(x,y);
  Delay(1);
until Ktb[ 1];

```

```

SetintVec($09,@oldKEY);

```

Asm

```

  mov ax,$0083
  int 10h
End;
END.

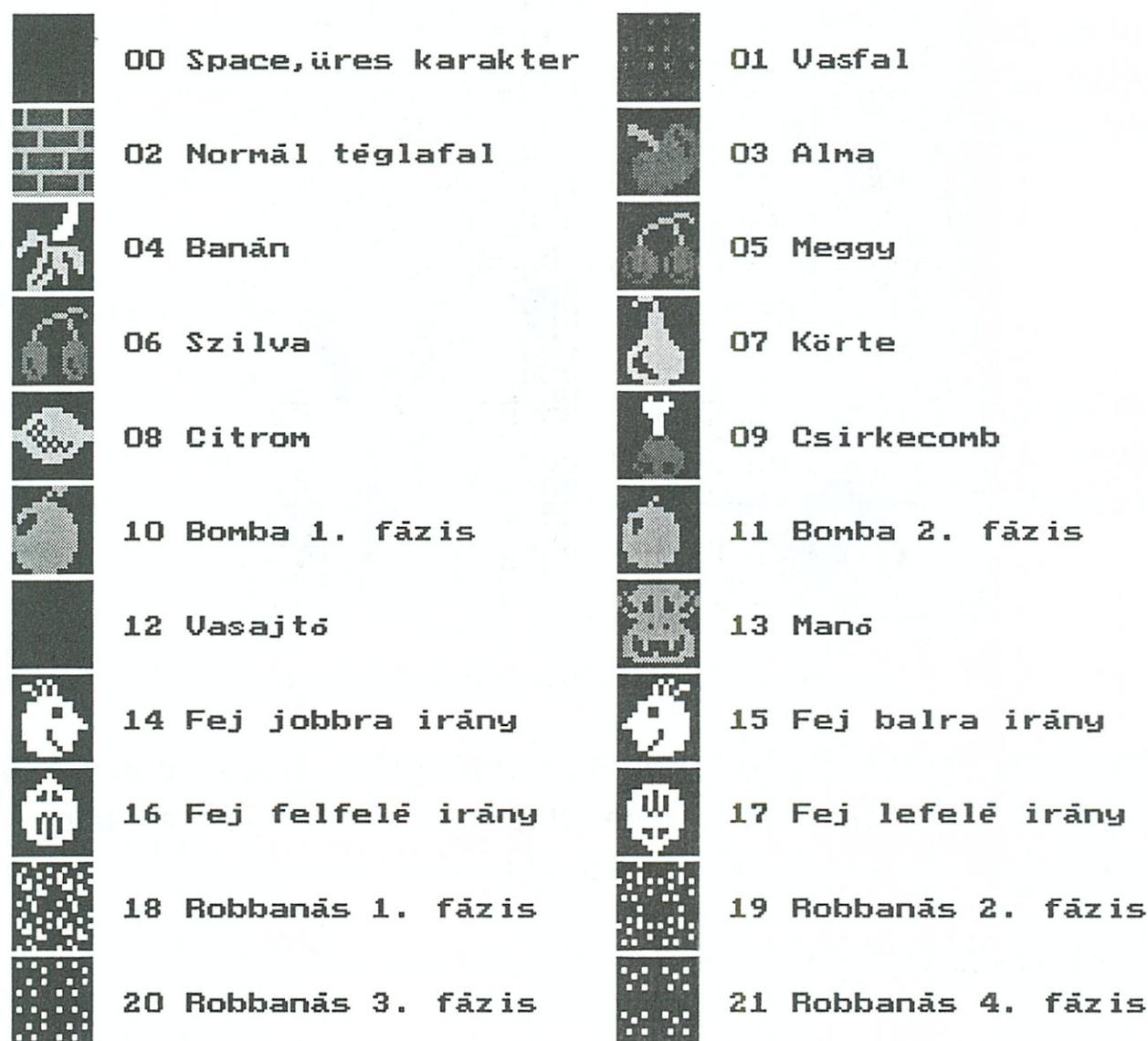
```

A program tulajdonképpen csak néhány sorban különbözik a pixelscroll programtól:

- A *ShowPic* rutin módosul egy Y korrigáló értékkel. Mivel a „B” kép mindig a 0000 címen kezdődik, ezért az „A” kép lejjebb kerül, mégpedig annyival, ahány soros a „B” kép.
- Normál *Writeln* utasítással írunk a „B” képre, hiszen ezt a Pascal úgy érzékeli, mintha a „B” normális 80 karakteres kép lenne.
- Az osztott képernyős kép installálásához 312-t töltünk a *Line Compare* regiszterbe, azaz ennél a rastersornál fog bekövetkezni a képosztás.

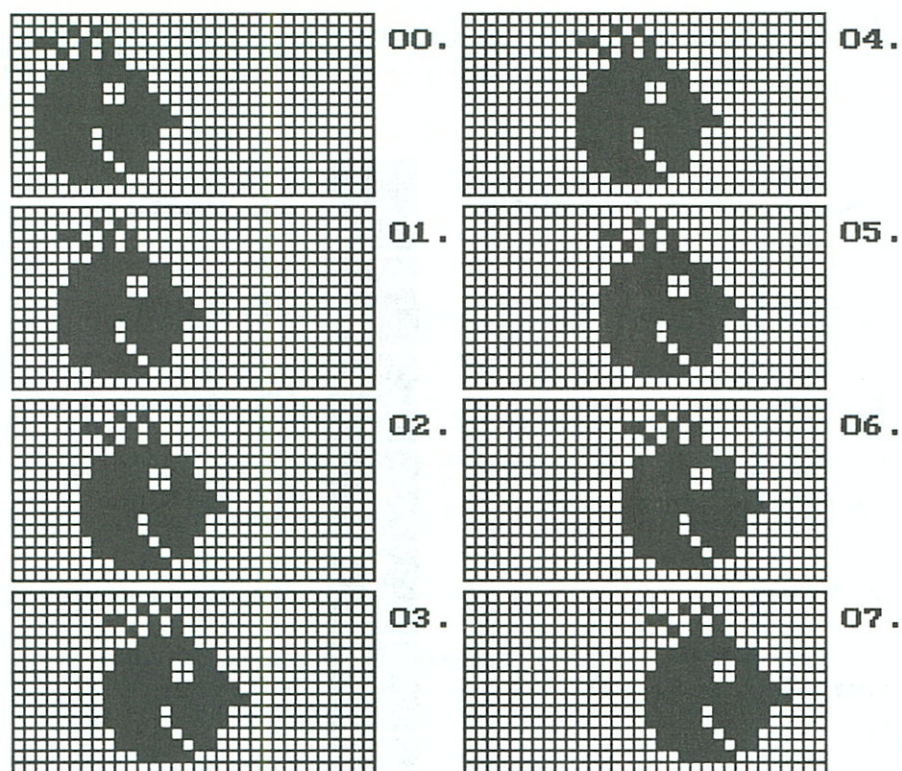
Az osztott képernyő bekapcsolása láthatóan igen egyszerű. A továbbiakban, a grafikus módoknál még hivatkozni fogunk az itt megismert fogalmakra (globális kép, ablak kép) és elvekre, hiszen ott is teljesen hasonlóan kell kezelni az osztott képernyős módot.

rekből a fenti kép BIG karaktereit rakjuk össze. Az összeállított játékelemeket a 23. ábra mutatja.



23. ábra A játék „BIG” karakterei

Ezeket a karaktereket a *BigChar* eljárás nyomtatja ki a felső képre. A program egyébként nem használja az összes itt szereplő karaktert. Ha elindítjuk a programot, akkor a figurát a Q, A, O és a P billentyűkkel mozgathatjuk. Látható, hogy a „fej” is pixelesen mozog. Ezt közvetlen memóriába írással oldottuk meg. A fej kétpixelenként lépked, és csak diszkrét (azaz nagy karakter határú) helyeken tud megállni. Mivel egy nagy karakter 16×16 pixeles, ezért a fejnek 8-at kell lépnie minden irányban, hogy végre tudjon hajtani egy teljes nagy karakteres elmozdulást. Ezeket a fázisokat előre megtervezve tároljuk a memóriában, ahonnan nyolc munkakarakterbe minden lépéskor kimásoljuk a megfelelő mozgásfázist. A mozgásfázisok közül a jobbra mozdulás elemei:



24. ábra A fej pixeles elmozdulása jobbra

Egy fázis 32×16 pixeles boxban fér el, azaz $4 \times 16 = 64$ bájt memóriát és 8 karaktert igényel. A többi irány fázisait is ugyanígy állítjuk elő, csak a függőleges irányok nem 32×16 -os, hanem 16×32 -es mezőben foglalnak helyet.

Az összes fázis memóriaigénye:

$$4(\text{irányszám}) * 8(\text{fázisszám}) * 8(\text{karakterszám}) * 8(\text{egy karakter bájtjai}) = 2048 \text{ bájt.}$$

A memóriabeli elhelyezkedés miatt különleges módon tároljuk a karaktereket: az első 8 bájt az első irány első fázisának első karakterét tartalmazza, a második 8 bájt a második irány első fázisának első karakterét és így tovább. Ezzel a tárolással tulajdonképpen leegyszerűsítettük a bemásolást, mert az egy karakterhez rendelt 32 bájt memória a 4 irányból egy-egy karaktert tartalmaz. Így a fázisváltásnál nem kell számolgatni, hanem csupán egy mutatót kell megnövelni 256-tal, és egy egyszerű **movsb** utasítással bemásolni a következő fázist. Az irányok között az eltolás így 8, míg az egyes irányok fázisai közötti eltolás 256. A módszert, amelyet „fésült” tárolásnak hívunk, máskor is fogjuk még használni, amikor grafikus elemek mozgásfázisait kell eltárolni.

A fenti készleteket a **gamechar.dat** és a **head.dat** fájlok tartalmazzák, de magában a programban már ezek *object* fájl változatait találjuk. Az *object* fájlokat a Pascal **BINOBJ** segédprogramjával állíthatjuk elő. A módszer előnye az, hogy a karaktera-

adatokat nem kell külön lemeztől beolvasni, hanem a programhoz szerkesztve a kód-szegmensben tároljuk azokat. A *BIG* karakterekhez a *Bchar* és a *Battr* tömbök adják meg az összerendelést. Minden *BIG*-hez megadjuk a hozzá tartozó négy normál karaktert és a négy színbájtót. Természetesen mindkét táblázat növelhető, így új karaktereket tudunk definiálni. Az alapkészletben a 170..177 karaktereket használjuk a fej pixeles megjelenítéséhez, de az ezek után következő karaktereket szabadon felül lehet írni, így növelni lehet a grafikai elemek számát.

5.9.2. Eljárások és függvények

A következőkben röviden ismertetjük a játék eljárásainak és függvényeinek működését. Talán azoknak sem haszontalan elolvasni ezt a részt, akik nem kívánják közelebbről megismerni a játékprogramot. Esetleg találhatnak valami érdekesebb algoritmust vagy programozási fogást.

```
(**      A billentyűzetet kezelő saját megszakítási rutin      **)  
Procedure KeyIRQ; Assembler;
```

A billentyűzet kezelése a fentebb már ismertetett elven történik. A *KeyIRQ* eljárás végzi a *KTB* logikai tömb dinamikus frissítését. Minden lenyomás és felengedés esetén megváltozik a megfelelő tömbelem. A játékot egyébként a Q, A, O és a P, valamint a szóköz billentyűkkel lehet vezérelni, míg az ESC lenyomására a program leáll. A mozgatási főciklusban az adott billentyűk természetesen szabadon megváltoztathatók.

```
(**      A játékképernyőt megjelenítő rutin (X,Y pixeles pozíció)      **)  
Procedure ShowPic; Assembler;
```

A *ShowPic* eljárás megjeleníti a játékterület, ugyanazon az elven, mint ahogy azt már a görgetés programozásánál leírtuk, csupán a rutin néhány numerikus értéke más a globális kép mérete miatt.

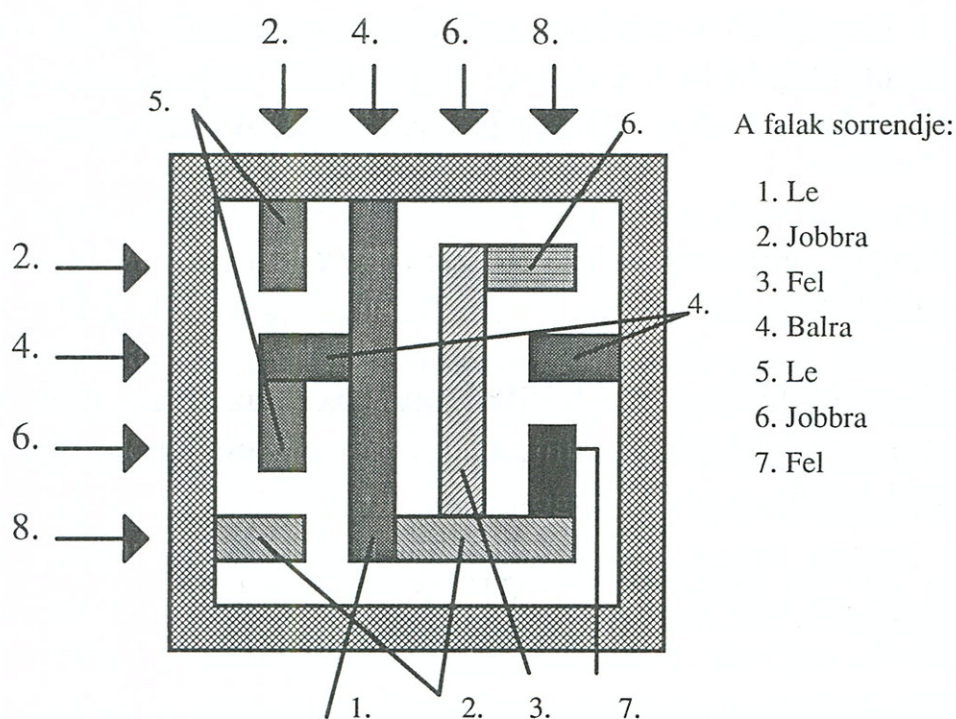
```
(**      Egy 2x2-es BigChar-t megjelenítő eljárás      **)  
Procedure BigChar(X,Y,C:Word); Assembler;
```

Az eljárás a játékban szereplő grafikus elemeket jeleníti meg a játékképen, a koordinátákat karakterpozícióban kell megadni. A rutin egyszerűen kiszámolja az adott koordinátához tartozó memóriacímet, és kinyomtatja a megfelelő karaktereket a megfelelő színekkel. A karaktereket a *Bchar* táblázatból, a színbájtókat pedig a *Battr* táblázatból veszi. Fontos, hogy a „B” kép miatti címetolódást nem az **offset**, hanem a **szegmens** értéken korrigáljuk, tehát az 5 karaktersoros eltolódást a képszegmens

\$B850-es értékével adjuk meg, így az „A” játékképnek és a „B” státusképnek is 0000 az offsetje. A karakter és a színtömb természetesen tetszőlegesen bővíthető, így bármilyen 2×2-es grafikus elemet elő lehet állítani. Például az alapkészletben szerepel a vasajtó karakter, és megvannak a mozgáshoz szükséges karakterek is.

(** A véletlenszerű labirintust generáló eljárás **)
Procedure Labirinth;

Az eljárás végzi a játékkép előállítását. A véletlenszerű labirintust generáló algoritmus régi és egyszerű. A labirintus az algoritmusnak köszönhetően teljesen bejárható, azaz minden pontjába el lehet jutni. A falak nem tartalmazzak szakadást, tehát a labirintus egy egyszerű balkéz szabállyal (minden elágazásnál balra kell fordulni) bejárható. Ebből az is következik, hogy a labirintus tetszőleges két pontját ki lehet jelölni be- és kijáratnak, mindenképpen ki lehet jutni belőle, de csak egyetlen kivezető út van. Hogyan végezzük a generálást? Tekintsük a labirintust egy egyszerű logikai mátrixnak! Ahol a mátrixban 1 (igaz) érték van, ott fal van, ahol pedig 0 (hamis), ott járható út. A labirintust fal veszi körül, azaz a mátrix szélei biztosan 1 értékűek. Fontos még az is, hogy a mátrix páratlan számú oszlopból és sorból álljon, hiszen logikus, hogy minden járatot két fal vesz körül, tehát n járat esetén $n+1$ fal szükséges ($n+n+1$ biztosan páratlan). Példaképpen nézzünk egy 11×11-es mátrixot:



25. ábra A labirintusgenerálás elve

Először a lehetséges falhelyekből véletlenszerűen kijelölünk egyet (most ez a 4. oszloppozíció), és húzunk egy falat fentről lefelé. A legfontosabb szabály, hogy amikor a fal egy merőleges fallal találkozik, akkor ott egy átjárót hagy ki. Az első fal után a

másodikat jobbról balra húzzuk szintén egy véletlenszerű helyen a lehetségesek közül (ez a 2. fal a 8. sorpozícióban). Látható, hogy itt is kihagytunk egy helyet, ahol a 2. fal az 1. fallal találkozott. Ez után a harmadik fal következik balról jobbra és így tovább. A labirintus akkor van készen, amikor vagy az összes vízszintes, vagy az összes függőleges helyről húztunk már falat. Ha a két méret megegyezik, akkor a kezdő fal helye megadja ezt, különböző méretek esetén természetesen a rövidebb oldal fog hamarabb megtelni. Végző soron az algoritmus nem tesz mást, mint minden alkalommal „megfelezi”, pontosabban szólva kettészeli a területet. A *LABIRINTH* eljárás is ezt végzi el egy 61×63-as mátrixon. Az algoritmus az egyenletes elosztás miatt sorban veszi az irányokat, de ezeket is lehet véletlenszerűen választani. Furcsa módon a labirintus „labirintusabb” így, ha mind a négy irány sorban következik, mivel nem fordulhat elő két azonos irányú fal egymás mellett.

```
(**           A 8 pixel széles üzemmódba kapcsoló eljárás           **)
Procedure Box8; Assembler;
```

Az eljárás a karakterszélességet váltja 8-pixelesre.

```
(**           A 40x30-as karakteres képernyőt bekapcsoló rutin           **)
Procedure Char40x30; Assembler;
```

Az eljárás bekapcsolja a 40×30-as karakteres módot. Ehhez a módhoz 480-ra kell növelni a függőleges sorok számát. A rutin működése szinte tökéletesen megegyezik a fejezet elején bemutatott módbeállító rutinok működésével. A függőleges felbontást az elektronsugár-vezérlő regiszterek közvetlen programozásával változtatjuk meg.

```
(**           Egy kis karaktert kirakó rutin (az alsó státusképre)           **)
Procedure LitChar(X,Y,C:Word); Assembler;
```

A *LitChar* eljárás az alsó képre az *X,Y* pozícióba kiírja a *C* szóval megadott karaktert és színt. A szürke keretező karaktereket is ezzel az eljárással írjuk ki. A *BigChar*-hoz hasonlóan ez az eljárás is kiszámolja az *X,Y* pozícióból az adott memóriacímet, és egy egyszerű *stosw* utasítással a *C* 16-bites szót kiteszi a megfelelő helyre.

```
(*           A saját TIMER megszakítás                               **)
Procedure TimIRQ; Assembler;
```

Az időzítő megszakítás rutinja. Ez a megszakítás gondoskodik a játékkép megjelenítéséről: a játékkép *X,Y* pozíciójából és a fej *Pox*, *Poy* pozíciójából kiszámítja, hogy a képnek éppen merre kell elmozdulnia. Az elmozdítást a *ShowPic* rutin meghívásával végzi el. Az eljárásban található egy inaktív *DELAY* hívás, amelyre esetleg szükség

lehet lassabb gépek esetén. Ha a képernyő görgetés közben megvillan, akkor a megjegyzésben lévő sort tegyük be a programba!

```
(**      A TIMER megszakítási intervallum beállítása                      **)
Procedure Settimer(p:word); Assembler;
```

Az eljárás az időzítő megszakítási időintervallumát állítja be. A PC-nek ezzel az eljárással kell megadni, hogy egy másodperc alatt hány megszakítást adjon. A megszakítási idő egyszerű osztással számítható ki:

$$\text{megszakítások száma} = \text{időzítő frekvencia} / \text{időzítő adatszó}$$

Alapesetben az időzítő adatszó \$0000, ami valójában 65 536-ot jelent, míg az időzítő működtető jel frekvenciája 1 193 180 Hz, így jön ki az alapesetben az $1\,193\,180 / 65\,536 = 18.206$ megszakítás másodpercenként. Például ha másodpercenként 100 megszakítást szeretnénk, akkor az időzítő adatszóba $1\,193\,180 / 100 = 11\,931$ -et kell írni. Így bármilyen megszakítási intervallumot be lehet állítani, de vigyázni kell, mert ha a megszakítások túl gyakoriak, akkor egymásba érhetnek, ez pedig lefagyáshoz vezethet.

```
(**      A fej pixeles eltolását megjelenítő rutin                      **)
Procedure PrintPhase; Assembler;
```

Ez a rutin a fej pixeles fázisainak a kinyomtatásáért felelős. A fázisszám 0 és 7 közé eshet, a 8. fázis már ismét a 0. képpel egyezik meg. A fázismozgatás a 170 és a 177 közé eső karakterek közvetlen átírásával történik. A karakterek tárolási formája miatt egyszerűen csak át kell másolni 256 (8*32) bájt a következő fázis beállításához.

```
(**      A pixeles elmozdulást elindító eljárás                          **)
Procedure StartMove; Assembler;
```

Az eljárás elindítja a fejet, azaz egy dupla karakteres pozícióból való induláskor beállítja a fázisváltozókat, és kirakja a 0. fázist.

```
(**      A pixeles elmozdulást folytató rutin                            **)
Procedure FlowMove; Assembler;
```

A fej pixeles elmozdítását végzi. Megnöveli a fázismutatót, majd a *PrintPhase* meghívásával kirakja a következő fázist. Ha a fázismutató eléri a nyolcat, akkor az előző pozícióba kirak egy SPACE (00) BIG karaktert, a fej pozíciójába pedig a megfelelő irányhoz tartozó fej karaktert (BIG 14..17). A program csak akkor kérdezi le a billen-

tyűket (tehát csak akkor lehetséges az irányváltás), ha a fej diszkrét pozícióban ($X, Y \bmod 16 = 0$) van.

```
(** GetNextChar - Az elmozdulás irányába eső BigChar kódját adja **)
Function GNC(ii:byte):Byte;
```

A *GNC* függvény lekérdezi a fej irányába eső következő *BIG* karaktert. A függvény a *BIG* bal felső normál karakteréből állapítja meg a *BIG* kódot, azaz nem teljesen helyesen, mert például a meggy és a szilva karakter között nem tud különbséget tenni.

```
(** A bombát felrobbantó eljárás **)
Procedure Robb;
```

A *ROBB* eljárás akkor aktiválódik, amikor egy bomba felrobban. Az eljárás a robbanási fázisokhoz tartozó *BIG* karaktereket nyomtatja ki.

Bár ezek az ismertetések nem mondhatók teljesen részletesnek és mélyrehatóknak, mégis hozzásegíthetnek a forráskód megértéséhez. A kísérletezéshez és a továbbfejlesztéshez sok sikert és sok türelmet (is) kívánunk!

5.10. A CrtPlus egység



CRTPLUS.PAS

A karakteres képernyő programozását már bemutattuk. A különböző témáknál ismertett eljárások gyakran hasznosak lehetnek, célszerű lehet beépíteni ezeket más programokba. A lemezmellékleten található **crtplus.pas** fájl egy olyan unit, amely a karakteres kép programozásához szükséges legfontosabb eljárásokat tartalmazza. Ezek sajnos hiányoznak a *crt* egységből. A fejezet befejezésekképpen vegyük sorra ennek a programnak a rutinjait és ezek működését. Az egységben néhány olyan rutin is szerepel, amelyről eddig még nem volt szó. Azok a részek viszont nem találhatók meg benne, amelyek túlságosan speciálisak ahhoz, hogy egy általános algoritmust lehessen alkalmazni (például osztott képernyős mód, pixeles scroll).

5.10.1. Az eljárások és a függvények csoportjai

Billentyűzetbővítések:

InstallKeyIRQ, UninstallKeyIRQ, (KeyIRQ)

Ablakkezelés:

FrameSize, GetFrame, PutFrame, CharChange, AttrChange, DrawBorder, ScrollUp, ScrollDn, ScrollLf, ScrollRg, SetCrtMode, NewWindow

Paletta- és színbeállítások:

Flash, SetEGAPalette, SetRGB

Karakterdefiníció:

DefChars, DefChar, Box8, Box9

Kurzorbeállítások:

CursorON, CursorOFF, SetCursorSize

Konverziók a számrendszerek között:

HexStr, BinStr, StrHex, StrBin

Egyéb:

SetTimer, EQ, NE

5.10.2. Változók, állandók

A *SetCrtMode* eljárásban használható állandók:

Const

```
_40x25 = 0;  
_40x28 = 1;  
_40x50 = 2;  
_80x25 = 3;  
_80x28 = 4;  
_80x30 = 5;  
_80x34 = 6;  
_80x50 = 7;  
_80x60 = 8;
```

A saját billentyűzetmegszakítás által kezelt logikai tömb:

KTB : **Array**[0..127] **of** Boolean;

A *SetCrtMode* és a *SetCursorSize* eljárásokkal beállított legutolsó értékek:

```
LastCrtMode : byte;      { Az utolsó beállított szöveges mód      }
LastCTop    : byte;
LastCBottom : byte;      { Az utolsó beállított kurzorméretek  }
LogicalSize : word;      { Az aktuális mód sorhosszúsága      }
```

A *SetCrtMode* eljárás által használt állandók módbeállításakor a **CRT** egység *windmax* változójába kerülnek:

```
Wmaxs : Array[ 0..8] of word=
($1827,$1b27,$3127,$184f,$1b4f,$1d4f,$214f,$314f,$3b4f);
```

A kurzor kezdő rasztersora egy-egy módhoz rendelve:

```
Curli : Array[ 0..8] of byte=(14,13,06,14,13,13,13,06,06);
```

Egy karaktersor logikai hossza (bájt/line) egy-egy módhoz rendelve: (\$3d4, \$13 regiszter)

```
LSize : Array[ 0..8] of word=(80,80,80,160,160,160,160,160,160);
```

A számkonvertáló eljárások által használt állandók:

```
Digit : Array[ 0..15] of char='0123456789ABCDEF';
Bits : Array[ 0..15] of word=
(32768,16384,8192,4096,2048,1024,512,256,128,64,32,16,8,4,2,1);
Hdig : Array[ 0..3] of word=(1,16,256,4096);
```

Az *InstallKeyIRQ* eljárás meghívásakor az eredeti rutin címének eltárolása:

```
OldKeyIRQ : procedure;      { Használja: Install-,
                               UninstallKEYIRQ }
```

5.10.3. Az eljárások és függvények működése

Minden ablakkezelő eljárásra érvényes, hogy a koordináták megadása nullával kezdődik. Ha például az alap 80×25-ös ablakot akarjuk beállítani, akkor ezt a *NewWindow(0,0,79,24)* utasítással tehetjük meg. Az eljárás egyébként azért készült, hogy a normál *Window* eljárás helyett lehessen használni, csak éppen az ésszerűbb és logikusabb számozással. A *NewWindow* mindazt elvégzi, amit a *Window*. Egyesek előtt ez talán illogikusnak tűnik, de a számítástechnikában általában a nullával kezd-

jük a sorszámozást (legalábbis a paraméterek megadásánál). Ugyancsak minden eljárásra érvényes, hogy a paraméterek helyességét egyik sem ellenőrzi, ezért a programozónak kell gondoskodnia arról, hogy illegális paraméter ne adódjék át. Az illegális paraméterek hibás működéshez, esetleg lefagyáshoz vezethetnek. Ablakdefiníció esetén tartsuk be az alábbiakat:

$X1 \leq X2$ és $Y1 \leq Y2$.

Eljárások:

Procedure AttrChange (*x1, y1, x2, y2*:word; *oldat, newat*:byte);

Az eljárás az *x1, y1, x2, y2* koordinátákkal megadott ablakban *newat* színbájtokra cseréli az *oldat* színbájtokat. A karaktereket változatlanul hagyja.

Procedure BinStr (*X*:word; **var** *S*:string);

Az eljárás bináris sztringgé alakítja az *X* 16-bites adatot. Az *S* sztring hosszúsága 16 karakter lesz, erre a programozónak kell ügyelnie a használatakor.

Procedure Box8;

Az eljárás 8 pixel széles módba kapcsolja a karakterkészletet.

Procedure Box9;

Az eljárás 9 pixel széles módba kapcsolja a karakterkészletet.

Procedure CharChange (*x1, y1, x2, y2*:word; *oldch, newch*:char);

Az eljárás az *x1, y1, x2, y2* koordinátákkal megadott ablakban *newch* karakterekre cseréli az *oldch* karaktereket, a színterületet változatlanul hagyja.

Procedure CursorOff;

Az eljárás kikapcsolja (eltünteti) a kurzort.

Procedure CursorOn;

Az eljárás bekapcsolja (ismét láthatóvá teszi) a kurzort. Ha a kurzor látható (ez az alapéset), akkor nincs hatása.

Procedure DefChar(dat:pointer; car, fm:word);

Az eljárás a *dat* mutató által mutatott adatokkal megadott mintára változtatja a *car* kódú karaktert, a karakter hosszát az *fm* paraméter adja. Az átírás közvetlen video-memóriába írással történik, az *fm* paraméter adja meg az átmásolandó bájtok számát, de nem változtatja meg az alapbeállítású karaktermagasságot. Az átírás villogásmentes.

Procedure DefChars(dat:pointer; fm:byte; car, db:word);

Az eljárással egyszerre több karakter is átírható. A *dat* mutató az új karaktermintákra mutasson, az *fm* paraméter a függőleges magasságot adja meg, a *car* a kezdőkarakter kódja, a *db* az átírandó karakterek darabszáma. Az átírás az *fm* paraméterrel megadott értékre állítja a függőleges karakterméretet. Az átírás villogással jár (mivel a BIOS karakterkészletet beállító rutinját használja), ezért célszerű üres képernyővel elvégezni.

Procedure DrawBorder(x1,y1,x2,y2:word; tp,col:byte);

Az eljárás az *x1,y1,x2,y2* koordinátákkal megadott karakteres keretet rajzol, amelynek típusát a *tp*, színét pedig a *col* paraméter adja. A típusok előre definiáltak, de felülírásuk lehetséges.

Function EQ(var x; var y; Size:Word):Boolean;

A függvény igaz értékkel tér vissza, ha a megadott *X* és *Y* típus nélküli változók egyenlőek. Az egyenlőséget a függvény a *Size-1* hosszon vizsgálja. Segítségével összetett adattípusokat tudunk összehasonlítani.

Program Osszehasonlitas;

Uses Crt, Crtplus;

Type Rec = **Record**

X, Y : word;

txt : **string**[40] ;

ptr : pointer;

End;

Var *A, B* : Rec;

Begin

```

A.X:=100;
A.Y:=200;
A.txt:='Szöveg';
A.ptr:=Nil;
B.X:=100;
B.Y:=200;
B.txt:='Szöveg';
B.ptr:=Nil;

if EQ(A,B,SizeOf(A)+1) then Writeln('A=B');

```

End.

A Pascal sajnos nem tud összetett adatokat összehasonlítani. Az *EQ* és a *NE* függvényekkel könnyen elvégezhetjük az összehasonlítást, nem kell elemenként összehasonlítani.

```
Procedure Flash(B:boolean);
```

Az eljárás a színbájtok 7. bitjének funkcióját állítja be. **True** esetén a 7. bit villogást jelent, míg **False** esetén 16 háttérszín használható.

```
Function FrameSize(x1,y1,x2,y2:Word):Word;
```

A függvény visszaadja az *x1,y1,x2,y2* koordinátákkal megadott ablak memóriaigényét. A függvény a *Getframe* és *Putframe* eljárással együtt karakteres ablakok eltárolására használható.

```
Procedure GetFrame(x1,y1,x2,y2:word; p:pointer);
```

Az eljárás a *P* mutató által mutatott helyre másolja az *x1,y1,x2,y2* koordinátákkal megadott ablakot. Használata előtt az ablak méretét a *FrameSize* függvénnyel állapíthatjuk meg, a memóriát pedig a *New*, *Getmem* eljárásokkal foglalhatjuk le.

```
Procedure HexStr(X:word; var S:string);
```

Az eljárás az *X* 16-bites adat hexadecimális formáját adja vissza az *S* sztringben. Az *S* sztring 4 karakter hosszú lesz.

```
Procedure InstallKeyIRQ;
```

Az eljárás bekapcsolja a módosított billentyűzetlekérdezést. A billentyűzet a bekapcsolás után az előző fejezetben leírtak szerint fog működni. Az eljárást csak egyszer szabad meghívni, mert különben a BIOS rutinra való visszatérés nem lesz lehetséges.


```
Function NE(var x; var y; Size:Word):Boolean;
```

A függvény összehasonlítja az *X* és *Y* változókat, és igaz értékkel tér vissza, ha a két változó nem egyenlő. Működése azonos az *EQ* függvényével (lásd ott).

```
Procedure NewWindow(x1,y1,x2,y2:integer);
```

Az eljárás a *Window CRT* eljáráshoz hasonlóan az *x1,y1,x2,y2* koordinátákkal megadott ablakot beállítja aktuális ablaknak. A koordinátákat a 0,0 ponttól kezdődően számozzuk! A *NewWindow* nem mozgatja a kurzort, ezért az utasítás után ajánlatos kiadni egy *CLRSCR* utasítást is!

```
Procedure PutFrame(x1,y1,x2,y2:word; p:pointer);
```

Az eljárás az *x1,y1,x2,y2* koordinátákkal megadott ablakba másolja a *p* mutató által mutatott memóriát. Az eljárás segítségével a *GetFrame* eljárással eltárolt ablak területe hozható vissza.

```
Procedure ScrollDn(x1,y1,x2,y2,e:word);
```

```
Procedure ScrollLf(x1,y1,x2,y2,e:word);
```

```
Procedure ScrollRg(x1,y1,x2,y2,e:word);
```

```
Procedure ScrollUp(x1,y1,x2,y2,e:word);
```

A *ScrollXX* eljárások egy karaktersorral/oszloppal a megadott irányba mozdítják az *x1,y1,x2,y2* koordinátákkal definiált ablak tartalmát. A bekerülő üres sort/oszlopot az *e* paraméterrel megadott karakterrel [lo(e)] és színnel [hi(e)] töltik fel.

```
Procedure SetCrtMode(mode:byte);
```

Az eljárás egy VGA karakteres módot állít be. A módok és az ezekhez tartozó előre definiált állandók a következők:

```
_40x25 = 0;
```

```
_40x28 = 1;
```

```
_40x50 = 2;
```

```
_80x25 = 3;
```

```
_80x28 = 4;
```

```
_80x30 = 5;
```

```
_80x34 = 6;
```

```
_80x50 = 7;
```

```
_80x60 = 8;
```

Ettől eltérő paramétert az eljárás nem ismer fel!

Procedure SetCursorSize (Top,Bottom:byte);

Az eljárás a kurzor méretét állítja be. A *Top* a kurzor felső, a *Bottom* a kurzor alsó rastersorának a számát adja meg.

Procedure SetEGAPalette (regnum,value:byte);

Az eljárás beállítja a *regnum* (0..15) számú elsődleges (EGA) palettaregisztert a *value* értékre. Az elsődleges palettaregiszterek mutatók a megfelelő RGB regiszterre.

Procedure SetRGB (palnum,Red,Green,Blue:byte);

Az eljárás a *palnum* (0..255) RGB regisztert beállítja a megadott *Red* (vörös), *Green* (zöld) és *Blue* (kék) értékre. A színtkomponensek hatbitesek.

Procedure SetTimer (p:word);

Az eljárás beállítja a timer megszakítások időintervallumát a *p* értékre. A timer 1 193 180/*p* darab megszakítást fog kérni egy másodperc alatt. Az alapesetet a *SetTimer*(0) utasítással állíthatjuk vissza.

Procedure StrBin (var s:string; var x:word);

Az eljárás az *X* 16-bites adatban adja vissza az *S* sztringben lévő bináris számot. Az *S* sztring legfeljebb 16 karakter lehet, és csak „0” és „1” karaktereket tartalmazhat. Más esetben az eljárás hibás adatot fog visszaadni.

Procedure StrHex (var s:string; var x:word);

Az eljárás az *X* 16-bites adatban adja vissza az *S* sztringben lévő hexadecimális számot. Az *S* sztring legfeljebb 4 karakter lehet, és csak a „0”...„9”, „a”...„f” és „A”...„F” karakterek szerepelhetnek benne. Más esetben az eljárás hibás adatot fog visszaadni.

Procedure UninstallKeyIRQ;

Az eljárás kikapcsolja a módosított billentyűzetkezelést, és visszaállítja az eredeti BIOS megszakítást. Csak akkor szabad alkalmazni, ha előtte már volt egy és csakis egy *InstallKeyIRQ* hívás. Más esetben a gép lefagy. Az *UninstallKeyIRQ* után az *InstallKeyIRQ* ismét használható, az egység ezt ismét az első hívásnak fogja értelmezni (lásd az *InstallKeyIRQ*-nál).

5.11. A CharEdit segédprogram



CHAREEDIT.PAS

A lemezmellékleten található segédprogrammal a 8×16 -os karakterkészletet tudjuk szerkeszteni. A programot nem ismertetjük, hiszen minden részét bemutattuk már. A program a kilépéskor a *char8x16.dat* fájlba kimentí a karakterkészletet, és bekapcsolva hagyja azt. Ha például a szerkesztés után belépünk a Pascal szerkesztőbe és az 80×25 -ös módban volt, akkor a saját karaktereinket használhatjuk. A program forrásprogramja természetesen tetszés szerint megváltoztatható és bővíthető.

Ezzel elérkeztünk a szöveges módokról szóló fejezet végére. Reméljük, hogy az itt közölt programokat, illetve elvi megoldásokat sokan tudják majd hasznosítani és továbbfejleszteni. Láthattuk, hogy a szöveges mód a VGA kártyákon hatékonyan programozható, és szokatlan feladatok megoldására is alkalmas (mint például az animáció). Nem állítjuk persze, hogy minden témát teljes részletességgel ismertettünk, de túlnyomó részben azért szó volt mindenről, amire a szöveges módokban lehetőség van.

6. A paletta programozása



6.1. Egy kis színelmélet

Az alapfogalmaknál már szó volt az additív és a szubsztraktív színkeverésről. Az életben általában a szubsztraktív keveréssel találkozunk gyakrabban. Mindenki kevert már különböző színű festékeket, és tudja, hogy minél több festéket keverünk, tehát minél több színt vonunk ki a fehérből, annál közelebb jutunk a fekete színhez. Ha viszont elvégezzük az 1. fejezetben leírt kísérletet, akkor láthatjuk, hogy a három alapszínből igenis kikeverhető a fehér szín. A kétféle színkeverés között az a különbség, hogy az egyik esetben a visszavert fényt látjuk, míg a másikban a kisugárzottat. Számunkra mindebből csak az a fontos, hogy a monitoron az R, G és B alapkomponeensekből minden színt ki tudunk keverni. Minden színkeverési probléma tehát arra vezethető vissza, hogy milyen arányban válasszuk meg az alapkomponeensek intenzitásait. Mivel általában a szubsztraktív keveréshez vagyunk szokva, ezért az RGB keverési módszerrel kapott eredmények némiképp meglepőek. Ki gondolná, hogy a piros és a zöld szín nem barnát (tehát a megszokottat), hanem sárgát fog eredményezni?

Nézzünk meg egy rövid példaprogramot, amely egyszerűen, mégis meggyőzően mutatja be az RGB színkeverést! A következőket fogjuk tenni: először kirajzolunk néhány függőleges piros vonalat a képre. A kép természetesen piros lesz. Ezt követően a piros vonalak mellé zöld vonalakat rajzolunk. Ha most távolról nézzük ezt a képet, akkor sárga összhatást kapunk, ugyanakkor közelről látható, hogy a képen csak piros és zöld vonalak vannak egymás mellett. Végül a piros és a zöld vonalak mellé kék vonalakat rajzolunk. Távolról ez egy fehér négyzetnek látszik, de közelről jól látható, hogyan válik külön a három alapszín.

Tehát a program:



RGBPELDA.PAS

```
Program RGB_szinkeveres_bemutatasa;
```

```
Uses Crt, Graph;
```

```
var i, j: integer;
```

```
BEGIN
```

```
  i:=Vga; j:=VgaHI;
```

```
  initgraph(i, j, 'c:\bp\bgi');      { Ide a saját elérési útvonal kell }
  { A 640x480/16-os mód beállítása }
```

```
  setfillstyle(1, red);               { Kirajzoljuk a piros csíkokat      }
```

```
  for i:=0 to 50 do
```

```
    for j:=0 to 50 do
```

```
      bar(j*6, i*2, j*6+1, i*6+1);
```

```
  Readln;      { Várakozás az ENTER-re }
```

```
  setfillstyle(1, green);             { Kirajzoljuk a zöld csíkokat      }
```

```
  for i:=0 to 50 do
```

```
    for j:=0 to 50 do
```

```
      bar(j*6+2, i*2, j*6+3, i*6+1);
```

```
  Readln;      { Várakozás az ENTER-re }
```

```
  setfillstyle(1, blue);              { Kirajzoljuk a kék csíkokat      }
```

```
  for i:=0 to 50 do
```

```
    for j:=0 to 50 do
```

```
      bar(j*6+4, i*2, j*6+5, i*6+1);
```

```
  Readln;      { Várakozás az ENTER-re }
```

```
  CloseGraph;
```

```
END.
```

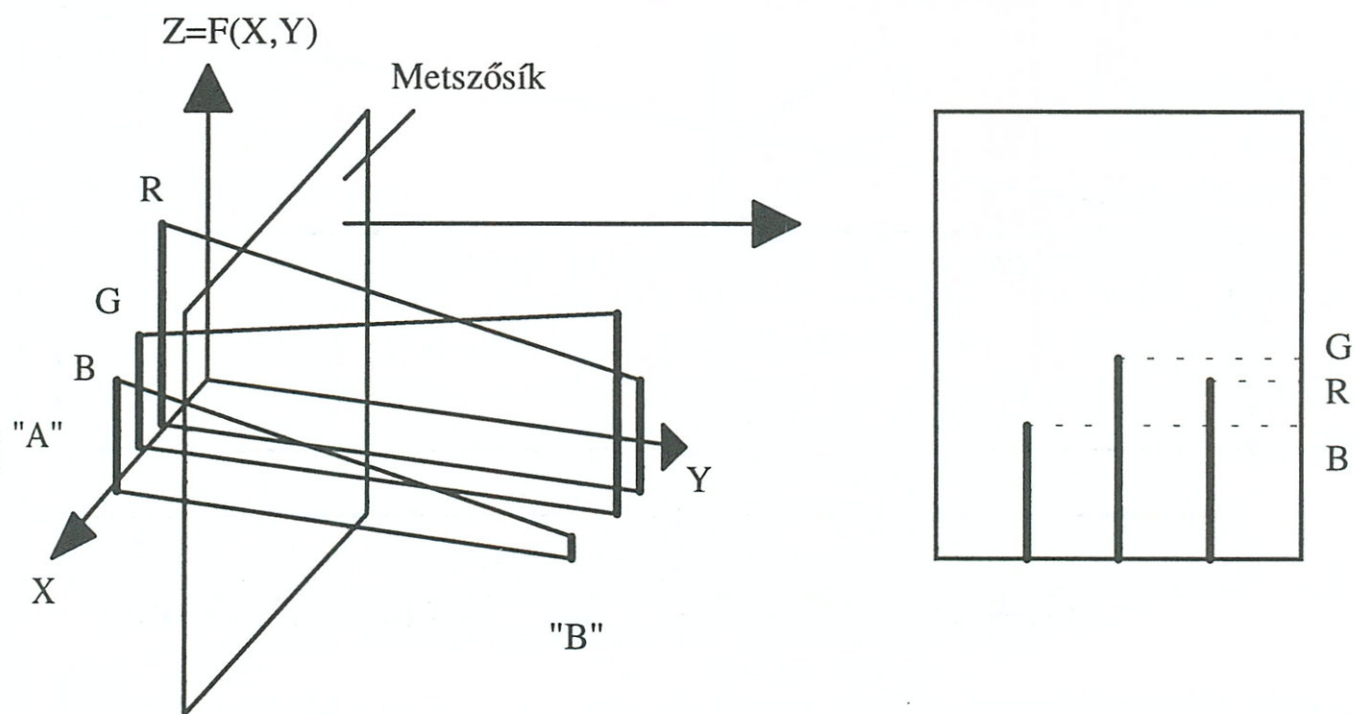
Bár a módszer szokatlan, és az eredmények meglepőek, azért ez a fajta keverés is megérthető és használható, csak egy kicsit bele kell jönni. A fejezet további részében erről a keverésről és a témával kapcsolatos lehetőségekről lesz szó.

6.2. A színkeverés alapelve

Ha színkeverésről beszélünk, akkor általában arra gondolunk, hogyan lehet kikeverni egy kezdő- és egy végszín között adott számú árnyalatot úgy, hogy az átmenet folyamatos legyen. A számítógépes keverés (a digitális technika miatt) lehetetlenné teszi a tökéletes folytonosságot, de megfelelően jó kvantálással (lépcsőkre bontással) a valósághoz közel álló, igen jó eredményeket tudunk elérni. Gondoljunk csak a True Color monitorkártyákra, ahol, bár a színkomponensek csak 8-bitesek, tehát elég durvának mondhatók, az előállítható színek száma 16 millió, ami az emberi szemnek már elég ahhoz, hogy (igencsak) színesnek érzékelje a képet. A VGA kártya esetében csak 6 bitet használhatunk fel egy-egy alapkomponensnél, de a színek száma már eléri a 262 ezret, tehát van miből válogatnunk.

Az árnyalatok keverésének problémáját nézzük a számítógép oldaláról:

Van egy „A” kiindulási színünk (amelynek a komponensei aR , aG és aB), és van egy „B” végérték színünk, amelynek a komponensei bR , bG és bB . Az „A” állapotból a „B” állapotba l lépésen keresztül kell eljutnunk. Szemléltessük ezt egy kétváltozós függvénnyel, amelynek egyik változója egy komponens (X), másik változója pedig a lépések száma (Y), míg a két függvény eredménye az adott szín $Z=F(X,Y)$.



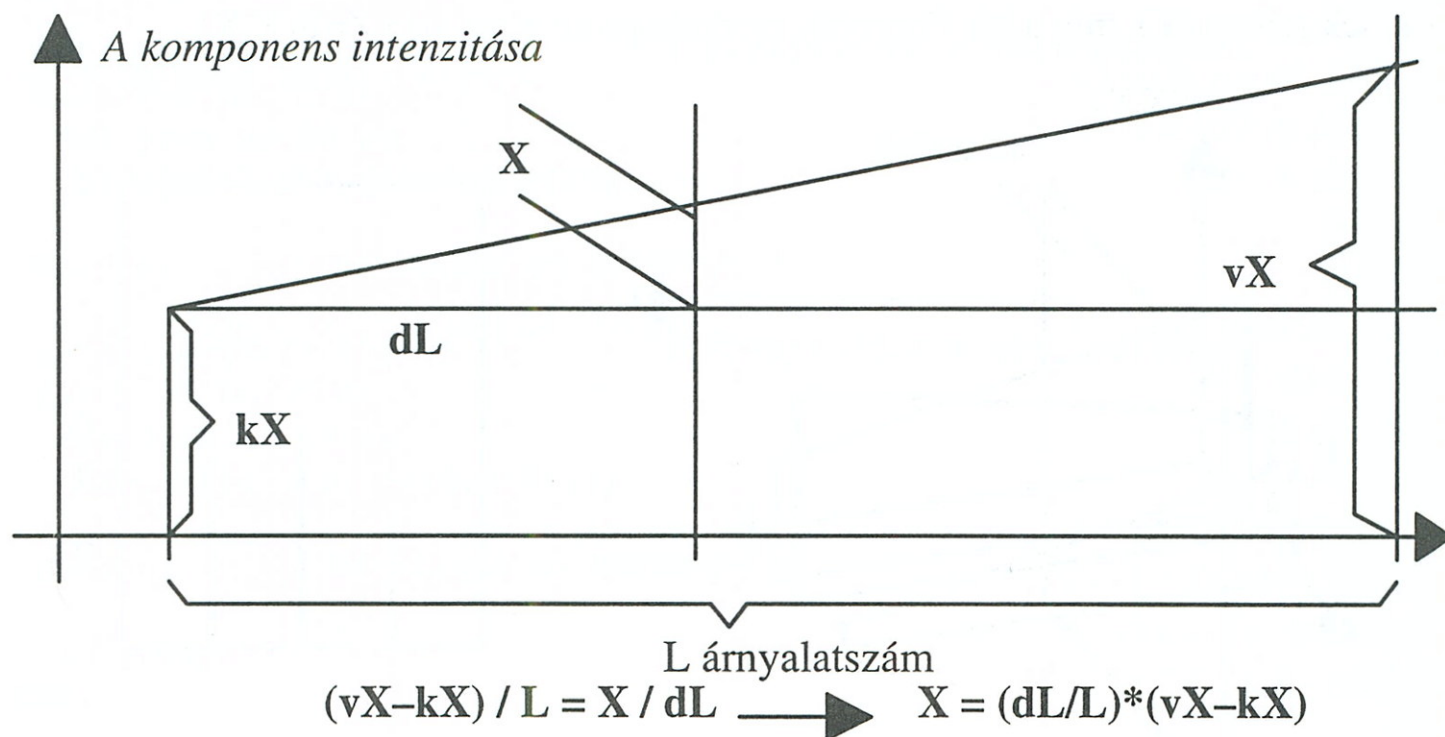
26. ábra A színkeverés ábrázolása

Az árnyalatok úgy határozhatók meg, hogy az Y hosszúságot felosztjuk l egyenlő részre, az egyeneseket elmetsszük az X — Z síkkal párhuzamos metszősíkokkal, és az így kapott helyi R , G és B színértékek adják ki az adott lépéshez tartozó árnyalat

színkomponenseit. Látható, hogy ebben az esetben egyenes vonalon jutottuk el a kezdőállapotból a végállapotba. Ez az ún. *lineáris színkeverési* módszer. Természetesen másfajta módon is elvégezhetjük a keverést, ezzel a közbenső árnyalatokat eltolhatjuk valamelyik irányba, például gyök- vagy hatványfüggvényekkel, és a három alapszínhez különböző függvényeket is rendelhetünk. Sok tehát a lehetőség, bátran lehet próbálkozni mindenféle matematikai boszorkánysággal. Az egyetlen, amire ügyelni kell, hogy az $F(X,Y)$ függvény értékei mindig a $[0,63]$ intervallumba essenek, más esetben ugyanis a szín hibás lesz! A következőkben egy lineáris és egy szögfüggvényekre alapuló színkeverést ismerünk meg.

6.3. A lineáris színkeverés

Az előző részben láthattuk, mit kell tennünk, hogy megkapjuk a megfelelő árnyalatokat. A feladat nem más, mint a három komponens kezdő- és végállapotát összekötni egy-egy vonallal, és kiszámolni a közbenső értékeket. Ezt a háromszög arányosságainak felhasználásával végezhetjük el a legegyszerűbben.



27. ábra A háromszög arányosság felhasználása

Ezzel az arányossággal kell dolgoznunk mindhárom alapszín esetében. A feladatot megvalósító program a következő:



RGBMIX.PAS

```
Program RGB_arnyalatok_keverese;
```

```
Uses Crt,Gr256; { Használjuk a saját egységünket is (később lesz szó róla) }
```

```
Type RGBreg = record      { Egy RGB adatrekord }
    r,g,b:byte;
end;
```

```
var
    kr,kg,kb,
    vr,vg,vb : byte;      { A kezdő- és a végállapot komponensei }
    i,j,k     : integer;
    Pal       : array[ 0..255] of RGBreg; { A paletta tárolása }
```

```
Procedure RGBline(l:byte);      { Lineáris keverési algoritmus }
Begin
```

```
    { A kezdő- és a végszint állítja be }
```

```
    pal[ l ].R:=kr;
```

```
    pal[ l ].G:=kg;
```

```
    pal[ l ].B:=kb;
```

```
    pal[ l ].R:=vr;
```

```
    pal[ l ].G:=vg;
```

```
    pal[ l ].B:=vb;
```

```
    { Lineáris összefüggéssel kiszámítja a közbenső árnyalatokat }
```

```
for i:=2 to l-1 do
```

```
    with pal[ i] do { A Pal tömb feltöltése }
```

```
    Begin
```

```
        if kr>vr then r:=kr+round((i-1)*(vr-kr)/(l-1))
```

```
            else r:=vr+round((l-i)*(kr-vr)/(l-1));
```

```
        if kg>vg then g:=kg+round((i-1)*(vg-kg)/(l-1))
```

```
            else g:=vg+round((l-i)*(kg-vg)/(l-1));
```

```
        if kb>vb then b:=kb+round((i-1)*(vb-kb)/(l-1))
```

```
            else b:=vb+round((l-i)*(kb-vb)/(l-1));
```

```
    End;
```

```
End;
```



```

{ -----}
BEGIN
  Screen_Init; { A gr256 egység eljárása (lásd a 256 színű módoknál) }
  { A 320x200/256-os módot állítja be }

  { Ezek a kezdőszín komponensei lineáris keveréskor }
  kr:=0;kg:=0;kb:=63; { kezdőszín: tiszta kék }

  { Ezek a befejező szín komponensei lineáris keveréskor }
  vr:=63;vg:=0;vb:=0; { végszín: tiszta piros }

  RGBline(160); { 160 árnyalatot kever ki }

  { A keverés eredményének megjelenítése }
  for i:=1 to 159 do
    Begin
      setRGBreg(i,pal[i].r,pal[i].g,pal[i].b);
      { gr256 egység: RGB beállítás }

      bar(i*2,10,i*2+1,190,i); { gr256 egység: a kitöltött
                                téglalap felrajzolása }

    End;

    outstrxy(0,192,'Press ENTER '); { gr256 egység: egy szöveg kiírása }
    readln; { Várakozás az ENTER lenyomására }

    textmode(c80);
  End.
{ -----}

```

A programban az *RGBline* eljárás végzi el az árnyalatok keverését, az *l* paraméter az árnyalatok számát adja meg. A kezdőszín komponenseit a *kr*, *kg* és *kb* változókkal, míg a végszín komponenseit a *vr*, *vg*, *vb* változókkal adjuk meg. Az eljárás használatakor ügyeljünk arra, hogy az *l* nagyobb legyen, mint kettő, hiszen ha csak két színt adunk meg, akkor az árnyalatok keverésének nincsen semmi értelme. A rutin az egyes komponensekre lépésenként elvégzi a számítást, és az eredményt a *Pal* tömbben helyezi el. A keverés után egyszerű *BAR* eljárással jelenítjük meg a kikevert színeket.

6.4. Árnyalatok szögfüggvények alapján

A következő eljárás három szögfüggvény alapján számítja ki a három komponenset. A piros színt a *SIN*, a zöldet *COS*, míg a kéket a $-SIN$ alapján számítjuk ki. A három komponens tehát úgy hat, mint ha fázisban eltolt szinuszhullámok lennének. Helyezzük el ezt az eljárást a fenti programban:

```

Procedure RGBsin(1:byte); { Keverés szögfüggvények alapján }
Begin
  for i:=1 to 1 do
    with pal[i] do { A Pal tömb feltöltése }
      Begin
        r:=32+round(30*sin( i*pi*2/1 )); { Piros : SIN függvényrel }
        g:=32+round(30*cos( i*pi*2/1 )); { Zöld : COS függvényrel }
        b:=32+round(30*sin( -i*pi*2/1 )); { Kék : -SIN függvényrel }
      End;
    End;

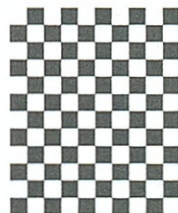
```

A programban az *RGBline(160)* utasítást cseréljük ki az *RGBsin(160)* utasításra! A futtatás eredményeképpen egy szivárványszíneket tartalmazó palettát kapunk.

6.5. Ha csak tizenhat színünk van...

Ekkor is lehet tenni valamit, hogy a kép színeiben valamivel gazdagabb legyen. A VGA kártya 262 144 színéből ekkor ugyan csak tizenhatot tudunk használni, de ezt a tizenhat színt mi magunk választhatjuk ki. Másrészt viszont, ha nagyobb területeket akarunk beszínezni, miért ne használhatnánk a legelemibb módszert, amelyet a monitor is alkalmaz?! Ha ugyanis meghatározott rendben különböző színű pixeleket rakunk egymás mellé, akkor ugyanazt tesszük, mint a monitor az elemi képpontokkal, csak nagyobb területen. Ez a módszer már régen ismert, és sok helyen alkalmazzák is. (Tipikus példa a Windows 16 színű módjai, ahol ilyen területfelosztásos módszerrel keverik ki a színeket.)

A legegyszerűbb ilyen módszer a **pepitamintás kitöltés (angolul fill)**, amikor két alapszínből egy újabb kevert színt állítunk elő az alábbi ábra szerint:



28. ábra Pepitamintás kitöltés

Ha az egyik szín a piros, a másik pedig a fehér, akkor eredményül rózsaszínt kapunk. A keverés persze csak megfelelően nagy felbontások esetén lehet sikeres, ahol az elemi pixelt már nehéz felismerni. A szabványos 640×480-as VGA felbontásban azonban már lehet alkalmazni, ezért nézzünk erre is egy programot!



COLORMIX.PAS

```

{$g+}
uses crt, graph;

var i, j, l: integer;

Procedure Box(x, y, a: word);           { Egy pepitamintás négyzet
                                         kirajzolása }

Begin
  for l:=0 to 15 do
    Begin
      setcolor(a and 15);               { Az első szín }
      line(x, y+l, x+16, y+l);
      setcolor(a shr 4);                { A második szín }
      line(x+1, y+l, x+16, y+l);
      asm
        rol byte [a], 4                 { A két szín felcserélése }
      end;
    End;
  End;

BEGIN
  i:=vga; j:=vgahi;                    { A 640x480/16-os videomódot
                                         használjuk }
  initgraph(i, j, 'c:\bp\bgi');        { Ide a saját útvonal kell }
  setlinestyle(4, $aaaa, 1);           { Ez a pepitaminta (101010101...) }

  for i:=0 to 15 do                    { A négyzetek kirajzolása }
    for j:=0 to 15 do
      if j>=i then Box(i*20, j*20, j*16+i);

  readln;                              { Várakozás az ENTER lenyomására }
  closegraph;

END.

```

Futtatás után a képen egy-egy négyzetben megjelennek a kikevert színek. Az átlóban a főszínek helyezkednek el. A módszert lehet bonyolítani is, például nem csak két színt használunk a keverésre, hanem többet, vagy másféle mintázatot alkalmazunk.

6.6. Animáció és mozgatás a paletta segítségével

A paletta és a színregiszterek alapvetően a színállapotok leírására szolgálnak, de amint azt sokszor tapasztalhattuk, az emberi agy mindenre ki tud találni valami mást is, mint amire azt készítői eredetileg szánták. A palettaregiszterekkel, közvetlen átírásukkal, a képernyőn a mozgás illúzióját tudjuk kelteni. Emlékezzünk vissza a szögfüggvényekkel létrehozott színsorra! Ha a képen látható szivárványábrát mozgatni akarnánk, akkor ez azt jelentené, hogy minden egyes függőleges vonalat (téglalapot) arrébb kellene mozdítani, tehát újra kellene rajzolni az egész képet. Ez az út még egy gyors géppel is nehezen járható, hiszen a 320×200/256-os kép 64 000 bájt a memóriában. Ha viszont a képet nem módosítanánk, hanem minden egyes színt egy palettaregiszterrel lejjebb másolnánk, akkor a hatás ugyanaz lenne, de csupán 3*256 OUT utasítást kellene végrehajtani. Írjunk tehát egy olyan eljárást, amely egy színnel elmozdítja, elforgatja a palettát. A 0. regisztert ne módosítsuk, mert az a háttér és a keret színe!

Procedure Palanim;

Begin

```
{ Először eltoljuk a színeket a MOVE utasítással }
move(colors,buff,3);           { Az első szín tárolása           }
move(colors[3],colors,762);    { Minden szín eggyel lejjebb lép }
move(buff,colors[762],3);      { Az elsőből lesz az utolsó    }
{ Az elmozgatott színeket kivisszük az RGB regiszterekbe }
```

Asm

```
cld
mov  dx,$3c8           { RGB írás címregiszter: $3C8      }
mov  al,1
out  dx,al             { Az 1. RGB regisztertől kezdve  }
lea  si,colors          { A DS:SI mutat az RGB értékekre }
inc  dx                { RGB adatregiszter: $3C9      }
mov  cx,255*3           { 255*3 bájtot küldünk ki        }
rep  outsb             { Az adatblokk kiküldése a $3C9    }
                           portra                                }
```

end;

End;

Az eljárásban a *colors* tömb tartalmazza az RGB értékeket, míg a *buff* egy hárombájtos segéd tároló, ahová mindig elmentjük a legelső színt. A **move** utasításokkal eggyel lejjebb mozgatjuk a színeket, majd a következő rövid Assembler rész az elmozdított színtáblázatot kiviszi az RGB regiszterekbe. Az RGB regisztereket természetesen

mindezek előtt fel kell tölteni a megfelelő színárnyalatokkal. Ezt követően már csak egy megfelelő képet kell készíteni, amelyen elvégezhető az animáció. A következő programban kétváltozós függvényeket ábrázolunk. Az ábrázolás a $Z=f(X,Y)$ összefüggésre épül, ahol X és Y pixelkoordinátaként, míg a Z színértékként jelentkezik majd a képen. A módszert úgy a legkönnyebb elképzelni, hogy a kétváltozós függvényt (amely egy felület) felülről nézzük, ahogyan egy térképet. A szintvonalakat a színek helyettesítik, de sokkal sűrűbbek és sokkal finomabbak az átmenetek. Az RGB regisztereket és a színtáblázatot a korábban ismertetett szögfüggvényes módszerrel töltjük fel.

Először nézzük a programot, azután pedig a részletes magyarázatot:



PALANIM.PAS

```
(*****)
(* Kétváltozós függvények ábrázolása síkban , palettaanimáció *)
(*****)
{ $G+}
Program Paletta_Animacio;

Uses Crt,CrtPlus;    { A Crtplus egység a lemez mellékleten található }

Var
    i,j,k,l : integer;
    x,y,z    : real;
    colors   : Array[ 0..764] of byte;  { Színtáblázat }
    buff     : Array[ 0..2]  of byte;

Procedure RGBsin(l:byte);                { Keverés szögfüggvények alapján }
var r,g,b:byte;
Begin
    j:=0;
    for i:=1 to l do
        Begin
            r:=32+round(30*sin( i*pi*2/l )); colors[ j] :=r;
            g:=32+round(30*cos( i*pi*2/l )); colors[ j+1] :=g;
            b:=32+round(30*sin( -i*pi*2/l )); colors[ j+2] :=b;
            SetRGB(i,r,g,b); inc(j,3);
        End;
    End;

    { Egy 256 színű pont kirakása }

Procedure Plot(X,Y:word;Z:byte);Assembler;
Asm
    mov ax,$a000
    mov es,ax                { Képszegmens }
    mov ax,320
```

```

mul    y
mov    di,x
add    di,ax           { Címszámítás: Cím = 320*Y + X }
mov    al,z
cmp    al,0
jnz    @x   { A 0. szint nem rakja ki, hanem 255-tel helyettesíti!! }
dec    al
@x:
    stosb      { A pixel kigyújtása }
End;

                                           { Az ábrákhoz tartozó függvények }

Function F(x,y:real;z:byte):Real;
Begin
    case z of
        0: f:=128+120*cos(x*pi)*cos(y*pi);
        1: f:=128+100*sin(x*pi/10)+100*sin(y*pi/10);
        2: f:=128+100*cos(x/pi*20)*sqr(y)+100*sqr(x)*cos(y/pi*20);
        3: f:=128+120*sqrt(abs(x)*pi)*sqrt(abs(y)*pi);
        4: f:=100+520*(x*x*y*y);
        5: f:=128+100*(20-abs(x))*cos(x*pi/20)*(20-
abs(y))*cos(y*pi/20)/4;
        6: f:=200*sin((pi)*sqrt(x*x+y*y));
    end;
End;

                                           { A 256 színű mód beállítása }

Procedure Vga256;Assembler;
Asm
    mov    ax,$13
    int    $10
End;

                                           { A paletta mozgatása }

Procedure Palanim;
Begin
    {... a fentebb ismerttetett eljárás ... }
End;

{ -----}
BEGIN
    Vga256;
    RgbSin(255);

    { Az adott függvény kirajzolása négyszeres tükrözéssel. Az összes
    függvény      }
    { szimmetrikus mind X, mind Y irányban, ezért elég egy negyedrészt ki-
    számolni      }
    For i:=-160 to 0 do
        For j:=-100 to 0 do
            Begin

```



```

k:=trunc(F(x+i*0.01,y+j*0.01,1));{ A függvény száma 0..6 lehet!!}
Plot(i+160,j+100,k);           { * <-- 0..6 }
Plot(159-i,j+100,k);
Plot(i+160,99-j, k);
Plot(159-i,99-j, k); { Tükrözések }
End;

repeat      { Palettaanimáció }
  palanim;
  delay(10);
until keypressed;

asm          { Kilépéskor visszaállítja a karakteres módot }
  mov  ax,3
  int  $10
end;
END.
{ -----}

```

Eljárások:

```
Procedure RGBsin(l:byte);
```

Ez az eljárás állítja elő azt a 255 színt, amelyet később mozgatunk.

```
Procedure Plot(X,Y:word;Z:byte); Assembler;
```

Az eljárás az X,Y pontba kirak egy Z színű pixelt, kivéve ha a Z=0, ekkor ezt 255-tel helyettesíti.

```
Function F(x,y:real;z:byte):Real;
```

A függvény megadja a hét különböző kétváltozós összefüggést, amelyek segítségével az ábrákat megjelenítjük. A függvények szimmetrikusak az origóra, ezért a kirajzolásnál csak egy negyedrészt számítunk ki, a másik három negyedképet tükrözéssel rajzoljuk ki.

```
Procedure Vga256; Assembler;
```

Az eljárás a VGA kártyát beállítja a 320×200-as felbontású, 256 színű grafikus módba.

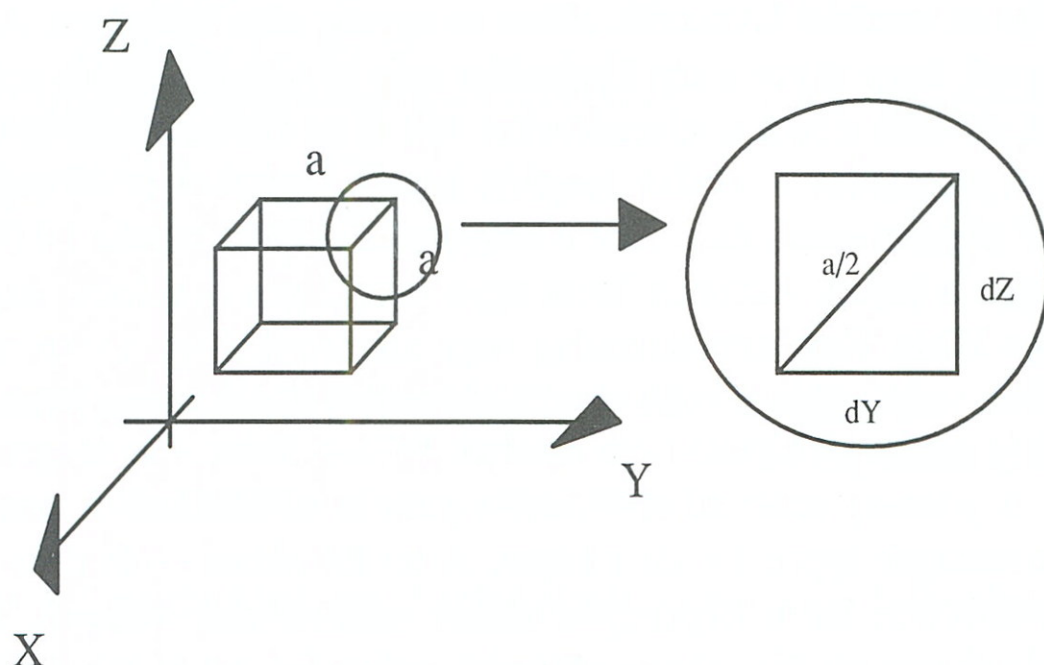
Procedure Palanim;

Az eljárás megegyezik az ismertetett palettamódosító eljárással, egy lépéssel lejjebb lépteti az összes RGB regisztert, ezáltal minden RGB regiszter a következő regiszter előző értékét kapja meg, míg a legutolsó regiszter az első regiszter előző értékét.

A főprogramban a módbeállítás és a színkeverés után a két **for** ciklus állítja elő a képet. A bal felső képnegyedet kiszámoljuk, a többi negyed pedig ennek tükrözött másolata. A kirajzolás után elindítjuk a palettarotációt, amitől a kép mozogni kezd. A **repeat until** belsejében lévő *delay* utasítással a mozgás sebességét szabályozhatjuk, míg az $F(X,Y,Z)$ függvényhívás Z paraméterével az előre definiált függvényekből választhatunk a $[0..6]$ intervallumban. Az F függvény tetszés szerinti függvényekkel bővíthető. A **Crtplus** egységet csak az *RGBsin* eljárásban lévő *SetRGB* utasítás miatt deklaráltuk, ha az utasítást kivesszük, akkor az egység nem szükséges. A különbség mindössze annyi, hogy ekkor a kép kirajzolása még az alapértelmezés szerinti palettával történik, és csak a rotáció elkezdésekor vált át az általunk beállított értékekre. Futtassuk le a programot, a látvány magáért beszél. Egyszerű problémát oldottunk meg egyszerű algoritmussal, mégis azt a benyomást keltjük, mintha 64 000 képpont izegne-mozogna a képen. Azonban (mert ugye semmi sem tökéletes) ez a program így alapvetően hibás. Nem szintaktikailag vagy szemantikailag, hiszen ezen a téren minden rendben van, csupán a VGA egyik kellemetlen tulajdonsága okoz némi szépséghibát. A kép szélén jól látható (már amelyik VGA-n látható) az idegesítő vibrálás és csíkozás. A jelenség azért áll elő, mert a paletta módosításakor nem figyeljük, hogy az elektronsugár éppen hol jár a képen. A továbbiakban — és itt is — ez most **NAGYON FONTOS: ha a képernyőn akkor módosítunk valamit, amikor az elektronsugár éppen ott jár, akkor villogást, csíkozást, tehát hibát okozunk.** Ez érvényes a palettaregiszterekre, de mint a későbbiekben látni fogjuk érvényes mindenféle tevékenységre, amelyet a képernyő látható részén végzünk. Ezért aztán már a videokártya-gyártók is azt a megoldást javasolják, hogy csak akkor módosítsunk az ilyen helyeken, ha az elektronsugár biztosan nem jár ott. Az elektronsugárra való várakozásról a karakteres kép programozásánál már volt szó. Ha el akarjuk kerülni a villogást, akkor itt is ezt kell tenni. A vertikális visszafutás ideje elegendő az összes regiszter átírásához, tehát nem fogunk villogást érzékelni. Csakhogy ez a várakozás tetemes időt elvesz a futási időből (a program mást sem tesz, mint vár), ezért a színmozgás lelassul. Kompromisszumot kell tehát találni a két véglet között. Egy lehetséges — ha nem is az egyetlen — megoldás az, hogy várakozunk a visszafutásra, de ilyenkor nem egy, hanem több színnel léptetjük tovább a palettát.

6.7. Axonometrikus ábrázolás és színrotáció

Fejlesszük tovább az előbbi programot! A színes mintákat előállító függvények kétváltozósak, azaz térgörbék. A fenti programban felülnézetből szemléltük ezt a térgörbét, és a magasságot a színekkel érzékeltettük. Egészítsük ki a függvény ábrázolását egy axonometrikus koordinátatranszformációval, és ábrázoljuk térben a függvényeket. Használhatnánk persze perspektivikus ábrázolást is, de az egyszerűség kedvéért most csak a 45 fokos axonometriával foglalkozunk. Ennek az ábrázolásmódnak az a szabálya, hogy 45 fokos dőléssel és a távolságok felezésével érzékeltetjük a térben tőlünk távolabb eső pontokat, tehát egy test vagy felület mélységét. Egy kocka axonometrikus képe a következő lesz:



29. ábra Az axonometrikus ábrázolás

A négyzet **X** irányú oldala $a/2$ hosszúságúra csökken, ezt a csökkenést kell a **dY** és **dZ** értékekre transzformálni, és így előállítani a képsík **I, J** koordinátáit. Pitagorasz tételével ezek az értékek könnyen kiszámíthatók. A transzformációs képletek a következők lesznek:

$$(X,Y,Z) \Rightarrow [I,J]$$

```
i := Round (xx0 + 160 + y - x / 2.8284);
j := 200 - Round (yy0 + (1/3) * 100 + (z/3) - x / 2.8284);
```

A képletekben szereplő konstansok:

160: a vízszintes felbontás fele (320/2)

100: a függőleges felbontás fele (200/2)

200: a kép $(0,0)$ pontját a bal alsó sarokba tesszük

2.8284: $2 \cdot \sqrt{2}$, azaz két gyök kettő az axonometrikus transzformáció miatt

Az $xx0$ és az $yy0$ a térbeli koordináta-rendszer origóját adja meg a képen. Az X , Y és Z értékek a térbeli koordináták, az I , J koordináta pedig az ezekhez tartozó pixel koordinátája a képen.

Ezeket az összefüggéseket felhasználva módosítsuk úgy az előző programot, hogy a függvényeket térben jelenítse meg. A térbeli ábrázolás miatt a függvényeket kis mértékben módosítanunk kell, hogy elférjenek a képen.

A módosított program:



PALANIM2.PAS

```
(***** )
(* Kétváltozós függvények ábrázolása axonometrikusan, palettaanimáció )
(***** )
{ $G+}
```

Program Paletta Animacio Terben;

```
Uses Crt,CrtPlus;
```

Var

```
i,j,k,l : integer;
```

```
x, y, z    : real;
```

```
colors : Array[ 0..764] of byte;
```

```
buff      : Array[ 0..2] of byte;
```

```
Procedure RGBsin(1:byte);           { Keverés szögfüggvények alapján }
```

```
var r,g,b:byte;
```

Begin

• • •

End;

{ Egy 256 színű pont kirajzolása }

```
Procedure Plot(X,Y:word;Z:byte); Assembler;
```

Asm

• • •

End;

{ Fekete függőleges vonal }

Procedure BVline(X,Y:word); **Assembler;**

Asm

```

    mov  ax,$a000
    mov  es,ax           { Az ES mutat a képre }
    mov  ax,320          { Címszámítás az X,Y alapján }
    mul  y
    mov  di,x
    add  di,ax           { DI = képcím }
    xor  al,al
    mov  cx,Y           { A ciklus fekete pontokat rajzol }
@cikl:
    mov  es:[di],al      { Egy fekete pont kirajzolása a képre }
    add  di,320          { A következő pont címe }
    inc  cx
    cmp  cx,200          { Egészen a képernyő aljáig rajzolunk }
    jnz  @cikl

```

End;

{ Koordinátatranszformáció }

Procedure Plot_2(x,y,z:integer);

var i,j:integer;

const xx0=-30;

yy0=50;

Begin

i:=Round(xx0+160+y-x/2.8284);

j:=200-Round(yy0+(1/3)*100+(z/3)-x/2.8284);

BVline(i,j); { Ha ezt kivesszük, akkor a kép alját is kirajzolja }

Plot(i,j,z);

End;

{ Az ábrához tartozó függvények }

Function F(x,y:real;z:byte):Real;

Begin

case z **of**

0: f:=128+120*cos(x*pi)*cos(y*pi);

1: f:=128+100*sin(x*x/pi*10)+100*sin(y*y/pi*10);

2: f:=128+100*cos(x/pi*20)*sqr(y)+100*sqr(x)*cos(y/pi*20);

3: f:=128+120*sqrt(abs(x)*pi)*sqrt(abs(y)*pi);

4: f:=100+520*(x*x*y*y);

5: f:=140+1.5*(20-abs(x))*cos(x*pi)*(20-abs(y))*cos(y*pi)/4;

6: f:=70+150*sin((pi)*sqrt(x*x+y*y));

end;

End;

{ A 256 színű mód bekapcsolása }

Procedure Vga256; **Assembler;**

Asm

...

End;

{ A színek mozgatása }

```

Procedure Palanim;
  Begin
    ...
  End;

{ -----}
BEGIN
  Vga256;
  RgbSin(255);
  For i:=-80 to 80 do
    For j:=-80 to 80 do
      Begin
        k:=trunc(F(x+i*0.01,y+j*0.01,0)); { A függvény száma:
                                           0..6 lehet!! }
        Plot_2(i+160,j+100,k);      { * <-- 0..6 }
      End;

    repeat
      palanim;      { Palettaanimáció }
    until keypressed;

  asm          { Kilépéskor visszaállítja a karakteres módot }
    mov ax,3
    int $10
  end;
END.
{ -----}

```

A program annyiban tér el az előző megoldástól, hogy teljesen kiszámítja a függvényeket, azaz nem használja ki a szimmetriát. Erre azért van szükség, hogy a képen megvalósítsuk a takarást. A takarást egyébként egyszerűen azzal oldjuk meg, hogy hátulról előre rajzoljuk fel a képet, és az újabb pontok felülírják a régieket. A *BVline* eljárás a felület alját tünteti el oly módon, hogy minden pontból egy függőleges fekete egyenest húz a kép aljáig. Ha az eljárást kivesszük a *Plot_2* eljárásból, akkor a térgörbék alja is látható lesz. A *Palanim* eljárás az előbbitől eltérően tartalmazza az elektronsugárra való várakozást, ezért a palettarotáció lényegesen lassabb, viszont a képen nincs szinkronizációs hiba. Ha a várakozó részt kivesszük, akkor a rotáció lényegesen felgyorsul, de a szinkronizációs hibák igen zavaróak lesznek.

6.8. A SetVGAPalette eljárás



PALSET.PAS

Az eljárást a paletta kézi beállítására készítettük, 640×480/16-os normál VGA módhoz. A lemez mellékleten a **PALSET.PAS** program tartalmazza az eljárást, amely szabadon beépíthető bármilyen programba. Az eljárás kinyit egy egyszerű grafikus ablakot, amelyben megjeleníti mind a 16 színt. A kurzorbillentyűkkel változtatni tudjuk a színeket. A le-fel nyilakkal kell kiválasztani, hogy színt vagy színkomponenst akarunk-e változtatni, míg a bal-jobb nyilakkal és a Home, End billentyűkkel a beállítást tudjuk elvégezni. Az ablak az ESC vagy az Enter lenyomására csukódik be, és ilyenkor visszaadja a vezérlést. Az ablak helyét meghíváskor az *X*, *Y* változóban adjuk meg (bal felső sarok), de arra vigyázni kell, hogy az ablak ne lógjon ki a képből. Az eljárás változói lokálisak, azaz nem igényelnek külön helyet az adatszegmensben, csupán a heapben foglalnak le helyet az ablak alatti terület tárolásához. A **PALSET** program elsőként véletlenszerűen kirajzol néhány vonalat a képre, majd a program egy billentyű lenyomásakor belép a színbeállításba. A beállítás után a program még egy billentyűlenyomást vár, mielőtt kilép. Mivel a program *readkey* függvénnyel kérdezi le a billentyűt, és a visszatérési értéket eldobja, ezért a programot bekapcsolt **X** direktívával kell fordítani ({**X**+}: eXtended syntax).

7. A tizenhat színű grafikus módok programozása



AG16MODE

A most következő fejezet a bitplane szerkezetű grafikus képek használatát mutatja be. A BIOS-szal beállítható szabványos bitplane szerkezetű üzemmódok a következők:

Mód (AL)	Felbontás
0Dh	320×200
0Eh	640×200
10h	640×350
12h	640×480

Látható, hogy a vízszintes felbontás 320, illetve 640 pixel, míg a függőleges felbontás 200, 350 és 480 lehet. Az utolsó fejezet, amely a nem szabványos, de beállítható módokat tárgyalja, megmutatja, hogyan lehet kombinálni a két vízszintes és a három függőleges értéket, és ezáltal többféle felbontást beállítani. A színek számát minden esetben 16-nak adják meg, ez természetesen azt jelenti, hogy ennél kevesebb színt is lehet használni. (A lehetséges értékeket a \$03C0, \$12 *Color Plane Enable* regiszterrel tudjuk beállítani.) A bitplane szerkezetet a képszerkezetek tárgyalásakor már megismertük, most megnézzük, hogyan lehet ezt hatékonyan programozni. Az alábbi témákra térünk ki:

- pixel kirajzolása és lekérdezése,
- vonalrajzoló algoritmus,
- a GrPlus egység: néhány hasznos rutin,
- mi is az a grafikus objektum, más néven a BOB,
- egyszerű (és buta) egyszínű BOB-megjelenítő,
- egy komplexebb egyszínű BOB-megjelenítő,
- többszínű, nem átfedő BOB-ok megjelenítése,
- többszínű, átfedő BOB-ok és megjelenítésük,
- BOB mozgásfázisok, BOB-animáció,
- osztott képernyő és görgetés,
- a képmemória tárolása és mozgatása.

Nos, ez egy hosszú fejezet lesz...

7.1. Egy pixel kirajzolása és lekérdezése

A Pascal grafikai egysége és a BGI állományok természetesen ellátják ezt a két feladatot. A most következő két rutint inkább az elv megismerése végett, semmint a gyakorlati felhasználásuk miatt mutatjuk be. (Egyébként gyorsabbak, mint a graph unit.)

7.1.1. Egy pixel kigyújtása

A feladat a következő: egy adott **X,Y** koordinátájú pontba rakjunk ki egy **Color** színű pixelt! Mit kell ehhez tennünk?

- kiszámolni az **X,Y** pozícióhoz tartozó képmemória bájt címét,
- a bájton belül kiszámolni a pixelhez tartozó bit pozícióját,
- rávenni a kártyát, hogy ebbe az egy bitpozícióba írja be a **Color** színt úgy, hogy a színes bájt más bitjeit ne módosítsa.

Az első feladathoz egy egyszerű képletet kell kiszámolnunk:

$$\text{Cím} = \text{ScrSize} * Y + X \text{ shr } 3,$$

ahol a **cím** a keresett memóriacím offsetje (a szegmenset ismerjük, \$A000), az **ScrSize** pedig az adott képernyő egy rastersorának logikai hossza. Az **ScrSize** a következő „ördögös” képlettel számítható ki:

$$\text{ScrSize} = \text{max.X felbontás} / 8$$

Ez 640-es felbontás esetén 80, 320-as felbontásban pedig 40. Ha valakinek az **ScrSize** valamilyen képernyőnél nem egész számra jön ki, akkor baj van. Az **X shr 3** azért szükséges, mert az **X** pozíció alsó 3 bitje a bitpozíciót adja, amelyet majd később használunk fel.

Tehát a címszámítás:

```

mov    ax,ScrSize    { A kép logikai szélessége           }
mul     Y             { Megszorozzuk az Y értékkel         }
mov     di,X          { DI-be tesszük az X értéket         }
shr     di,3          { Az X alsó 3 bitje most nem kell     }
add     di,ax         { A DI-ben most megvan a cím          }
```

A bitpozíciót meghatározni egy tömb segítségével egyszerűbb, mint forgatással:

```

const Bits : Array[ 0..7] of byte=(128,64,32,16,8,4,2,1)

lea    bx,Bits           { A pozíciótömb címe a BX-be }
mov    al,byte ptr [X] { Az X alsó nyolc bitje az AL-be }
and    al,7              { Csak az X alsó 3 bitje számít }
xlat                   { AL most a bitmaszkt tartalmazza }

```

A pixelnek most már ismerjük a bájt- és a bitpozícióját. A kigyújtáshoz válasszuk a **mode 2** írás módot. Használhatnánk az egyes módot is, de azzal a kigyújtás sokkal körülményesebb lenne. Hogyan működik a kettes írás mód? A \$3CE, \$08 bitmaszk regiszterrel meg kell adnunk, hogy melyik bitet akarjuk módosítani, és a képbájtba ki kell vinnünk a színértéket. A kivitel előtt egyszer be kell olvasni a képbájtot, hogy a VGA latch feltöltődjön. Ha ezt nem tennénk meg, akkor a bájt nem érintett bitjei törlődnének vagy módosulnának.

```

{ Feltétel: A kártya Mode 2 írásmódban van, }
{ és az ES az A000h szegmensre mutat }
mov    dx,03ceh          { A bitmaszk regiszter I/O címe }
mov    ah,al              { Az előző bitpozíció }
mov    al,08              { A bitmaszk regiszter a 8-as }
out    dx,ax              { Kiküldjük a bitmaszkt }
mov    al,ES:[di]         { A képmemória olvasása, a latch feltöltése }
mov    al,Color           { A színt az AL regiszterbe tesszük }
stosb                    { A pixel most megjelenik a képen }

```

Tehát a teljes, összerakott eljárás a következő lesz:

```

{ Állandók }
Const
    SegA000 : word = $a000; { A 7.0-s Pascalban ez előre definiált }
    Bits    : array[ 0..7] of byte=($80,$40,$20,$10,$08,$04,$02,$01);
    ScrSize = 80;

```

Procedure Pixel(X,Y,Color:Word); **Assembler;**

Asm

```

lea    bx,Bits           { A BX adja majd meg a megfelelő bitet }
mov    es,SegA000        { Az ES a képernyőre mutat }
mov    ax,ScrSize         { Címszámítás: }
mul    Y                  { Cím := 80*Y + X shr 3, }
mov    di,X               { ahol a 80 a kép szélessége bájtokban }
shr    di,3               { Az X alsó három bitje nem kell }
add    di,ax              { ES:DI mutat a képbájtra }
mov    al,byte ptr [X]    { X and 7 megadja a bitszámot }
and    al,7
xlat                    { Az AL-be a bitmaszk }
mov    ah,al              { Onnan pedig az AH-ba }

```



```

mov     al,08
mov     dx,03ceh      { $3Ce, $08 : bitmaszk regiszter      }
out     dx,ax          { A maszkregiszterrel kiválasztjuk      }
                        { a bitet                               }

mov     al,es:[ di]    { Ez az olvasás tölti fel a VGA      }
                        { latch-et                               }

mov     al,byte ptr [ Color] { Ez lesz a pixel színe          }
stosb                                { A latch és az adat összekombinálása }
                        { és kivitele a memóriába: a pixel    }
                        { megjelenik                           }

End;

```

7.1.2. Egy pixel színének lekérdezése

A feladat most az, hogy meghatározzuk az adott **X,Y** pozícióban lévő pixel színét. Erre a **Mode 0** olvasás módot fogjuk használni. Ez az olvasási mód egyetlen bitplane-t olvas le, tehát mind a négy plane-t le kell kérdeznünk, és így kell összeállítanunk a négybites eredményt. A cím kiszámításáig és a bitpozíció meghatározásáig ugyanazt kell tennünk, mint a kigyújtásnál. Utána egy olvasóciklussal minden bitplane-ről le kell olvasni a megfelelő bitet.

```

mov     bl,al          { A BL adja a bitmaszkot              }
xor     bh,bh          { BH-ban jelenik meg az eredmény      }
mov     ch,8           { CH a bithelyirértékeket adja majd    }

mov     dx,03ceh
mov     ax,0304h       { Az olvasásregiszter a 3. plane-re mutat }

@getc:
out     dx,ax          { Az olvasandó plane kiválasztása      }
test    es:[ di] ,bl    { Az adott plane-en a bit megvizsgálása }
jz      @nobit         { Ugrás, ha a bit nulla              }

@bit:
or      bh,ch          { Ha a bit 1, akkor az eredménybe 1-et tesz }

@nobit:
shr     ch,1           { A következő bit, a CH elforgatása    }
dec     ah             { A következő plane                    }
jns     @getc          { 4 plane-t vizsgál                    }
mov     al,bh          { A visszatérési értéket az AL-be kell tenni }

```

A leolvasást a 3. plane-nel kezdtük el, a bit értékét a **TEST** utasítással vizsgáltuk. Az eredmény a **bh** regiszterben képződött. A teljes függvény a következő:

```
Function GetPixel(X,Y:Word):byte; Assembler;
```

```
Asm
```

```
    lea    bx,Bits
    mov    es,SegA000
    mov    ax,ScrSize      { A pixelnél látott címszámítás }
    mul    Y
    mov    di,X
    shr    di,3
    add    di,ax           { Az ES:DI a képbájtra mutat }
    mov    al,byte ptr [X]
    and    al,7
    xlat
```

```
{ Innentől a fenti programrészlet következik. }
```

```
End;
```

A két rutin működését a **PIXELRUT.PAS** program mutatja be:



PIXELRUT.PAS

```
{ $G+ }
```

```
Const SegA000 : word = $a000; { A 7.0-s Pascalban ez előre definiált }
      Bits      : array[ 0..7] of
byte=($80,$40,$20,$10,$08,$04,$02,$01);
      ScrSize = 80;
```

```
Var
```

```
    i,j      : word;
```

```
{ Az írás mód beállítása }
```

```
Procedure WriteMode(Mode:byte); Assembler;
```

```
Asm
```

```
    mov    dx,03ceh { Az írás mód beállítása a $3ce, $05 regiszterrel }
    mov    ah,Mode
    mov    al,05
    out    dx,ax
```

```
End;
```

```
{ Egy pixel kirakása }
```

```
Procedure Pixel(X,Y,Color:Word); Assembler;
```

```
Asm
```

```
    { A fenti pixel eljárás }
```

```
End;
```



```

{ Egy pixel színének lekérdezése }
Function GetPixel(X,Y:Word):byte; Assembler;
  Asm
    { A fenti getpixel függvény }
  End;

BEGIN
  begin
    mov  ax,0012h { 640x480/16 színű mód }
    int  10h
  end;

  Writemode(2); { A pixel rutinhoz 2-es írás mód szükséges }

  { Véletlenszerűen pixeleket rak ki }
  for i:=0 to 65535 do
    Pixel(random(640),random(480),random(16));

  { Pixelenként invertálja a képet (negatív kép) }
  for i:=0 to 479 do
    for j:=0 to 639 do
      Pixel(j,i,15-GetPixel(j,i));

  { Pixelenként ismét invertálja a képet (megint pozitív kép) }
  for i:=0 to 639 do
    for j:=0 to 479 do
      Pixel(i,j,15-GetPixel(i,j));

  Writemode(0);
  portw[$3ce] := $ff08; { A maszkregisztert inicializálni kell a kiírás
                        miatt }

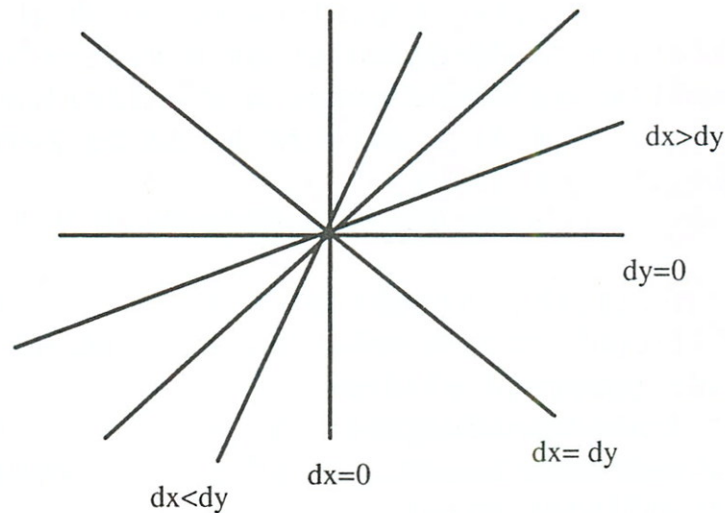
  write('Press ENTER');
  { Vár az ENTER lenyomására }
  readln;
  asm
    mov  ax,0003h          { 80x25 karakteres mód }
    int  10h
  end;
END.

```

7.2. Vonalarajzoló algoritmus

A klasszikus feladatmegadás az, hogy egy adott $X1, Y1$ kezdő- és $X2, Y2$ végpont között húzni kell egy lehetőleg egyenletes vonalat. Az egyenletesség a lépcsőzetességre vonatkozik, hiszen tudjuk, hogy a kép kis téglalap alakú pixelekből épül fel, tehát alapvetően diszkrét elemekből; míg egy vonal elemi pontjai nem biztosan esnek

egész pixelhelyekre. A kezdőpont és a végpont elhelyezkedéséből adódóan nézzük meg a különböző eseteket!



30. ábra Vonalrajzolás

Ha $dx=0$, akkor egy egyszerű függőleges vonalat kell húznunk, ha $dy=0$, akkor a vonal vízszintes, ha pedig $dx=dy$, akkor a vonal 45 fokos. Ebben a három esetben a vonal pontjai egész pixelhelyekre esnek, tehát nincs szükség arányos elosztásra. Azonban a $dx>dy$ vagy a $dy>dx$ esetben valamilyen ferde vonalról van szó, és ekkor már kis, arányosan elosztott részekre kell bontani a vonalat. A $dx>dy$ esetben az Y irányú elmozdulás egyesével lépked, hiszen $dy<dx$, és a legkisebb lépés 1 pixel, míg az X elmozdulás dx/dy arányban felosztva csak minden i -edik lépésnél fog nőni, ahol a tulajdonképpeni feladat az i értékének meghatározása. Mindezek alapján a vonalhúzást több kisebb részrutinra lehet bontani:

1. Ha $dx=0$, akkor függőleges vonalat kell húzni.
2. Ha $dy=0$, akkor vízszintes vonalat kell húzni.
3. Ha $dy \geq dx$, akkor Y léptékű arányos ferde vonalat kell húzni.
4. Ha $dx > dy$, akkor X léptékű arányos ferde vonalat kell húzni.

A 3. pontban a 45 fokos vonal húzása is szerepel! A most következő példa az algoritmus magas szintű megvalósítását mutatja be. Az eljárás a fenti vizsgálatok alapján a megfelelő részrutinnal elvégzi a kirajzolást. A **Graph** egységben található *Line* eljárás egyébként ugyanezen az elven rajzolja meg a vonalat.



LINERUT.PAS

Uses graph;

Var

```
i, j, k, l: integer;
x, y, x0, y0: integer;
```


{ Két egész típusú változó értékének cseréje }

Procedure Swap(**var** a,b:integer); **Assembler;**

Asm

```

push DS
lds  si,a           { DS:SI mutat az A változóra      }
les  di,b           { ES:DI mutat a B változóra      }
mov  ax,[si]        { Az A változó az AX-be kerül     }
xchg ax,es:[di]     { Csere                           }
mov  [si],ax        { Az AX visszakerül              }
pop  DS

```

End;

{ A LINE algoritmust bemutató eljárás

A line rajzolását 4 esetre bontja:

dy=0 , ekkor vízszintes vonal

dx=0 , ekkor függőleges vonal

dx>dy , X növekményes ferde vonal

dy>=dx , Y növekményes ferde vonal (és a 45 fokos vonal)

}

Procedure Lline(x1,y1,x2,y2,color:integer);

Var dx,dy,n,m:integer;

Begin

dx:=abs(x2-x1); { X távolság } }

dy:=abs(y2-y1); { Y távolság } }

Putpixel(x1,y1,color); { Kirajzoljuk a kezdőpontot } }

{ Ez a kezdőpont }

if y1<y2 **then** n:=1 **else** n:=-1; { A függőleges lépték az n }
változó }

{ Az n a függőleges léptéket adja (SGN Y) }

if x1<x2 **then** m:=1 **else** m:=-1; { A vízszintes lépték az m }
változó }

{ Az m a vízszintes léptéket adja (SGN X) }

if (dx=0) **and** (dy=0) **then** exit;

{ Ha a kezdőpont megegyezik a végponttal, akkor kilép } }

if dy=0 **then** { Vízszintes vonal }

Begin

if x1>x2 **then** swap(x1,x2); { Csere, ha X1>X2 } }

for i:=1 **to** dx **do** Putpixel(x1+i,y1,color);
{ A vonal kirajzolása } }

exit; { Kilépés az eljárásból } }

End;

```

if dx=0 then { Függőleges vonal }
  Begin
    if y1>y2 then swap(y1,y2); { Csere, ha Y1>Y2 }
    for i:=1 to dy do Putpixel(x1,y1+i,color);
                                { A vonal kirajzolása }
    exit;                      { Kilépés az eljárásból }
  End;

```

```

if dx>dy then { X növekményes ferde vonal }
  Begin
    i:=dx shr 1;                { A növekmény előállítása INT (X/2) }
    for l:=1 to dx do          { DX számú pontot fogunk kirakni }
      Begin
        inc(i,dy);              { Az i változó növelése }
        if dx<=i then          { Ha az i>dx, akkor átlós lépés jön }
          Begin
            dec(i,dx);          { Átlós lépésnél az i-t korrigálni kell }
            inc(y1,n);          { Az átlós lépés ilyenkor Y irányú }
          End;
        inc(x1,m);              { A főirányú lépés ilyenkor X irányú }
        Putpixel(x1,y1,color);  { A lépés megjelenítése }
      End;
    exit;                      { Kilépés, ha kirajzoltuk a vonalat }
  End;

```

```

if dy>=dx then { Y növekményes ferde vonal }
  Begin
    i:=dy shr 1;                { A növekmény előállítása INT(Y/2) }
    for l:=1 to dy do          { Most DY számú pontot kell kirajzolni }
      Begin
        inc(i,dx);              { Az i növekmény növelése }
        if dy<=i then          { Ha az i>dy, akkor átlós lépés }
          { következik }
        Begin
          dec(i,dy);            { Átlós lépésnél korrigáljuk az i-t }
          inc(x1,m);            { Az átlós lépés ilyenkor X irányú }
        End;
        inc(y1,n);              { A főirányú lépés ilyenkor Y irányú }
        Putpixel(x1,y1,color);  { A lépés megjelenítése, pontrajzolás }
      End;
    exit;                      { Kilépés, ha a vonal készen van }
  End;
End;

```



```

(***** A főprogram kezdete *****)
Begin
  i:=Vga;j:=VgaHi;
  initgraph(i,j,'c:\bp\bgi');{ Ide a saját BGI elérési útvonalunkat
kell írni }
  { A 640x480/16-os grafikus mód bekapcsolása }

  { A kép közepéből egy ellipszis kerületéhez húz vonalakat }
  for k:=0 to 360 do { Egy teljes szög minden fokát vesszük }
    Begin
      x:=320+round(300*sin(k*pi/180)); { Az X végpont }
      y:=240+round(200*cos(k*pi/180)); { Az Y végpont }
      Lline(320,240,x,y,k and 15); { A vonalhúzó algoritmust
                                     hívja }
    End;

  write('Press ENTER'); { Kilépéskor az ENTER-t kell lenyomni }
  readln;
  closegraph; { A grafika lezárása, a heap
               felszabadítása }

End.

```

A rutin könnyebb megértése érdekében az egyes vonalak megrajzolásakor célszerű lépésről lépésre követni ezt a programot a beépített *debuggerrel*. A vonalhúzásról bővebben még a 256 színű mód tárgyalásakor is lesz szó, ott megismerjük majd a fenti algoritmus assembly megvalósítását is.

7.3. A GrPlus egység



GRPLUS.PAS

Ez az egység olyan eljárásokat tartalmaz, amelyekre általában gyakrabban van szükség a grafikus programokban. A legfontosabb ilyen feladatok:

- pixel kigyújtása és lekérdezése,
- vízszintes és függőleges vonalhúzás,
- karakter kiírása,
- üres téglalap rajzolása,
- kitöltött téglalap rajzolása,
- térhatású téglalap rajzolása,
- képernyőtörlés.

Tegyük egy kis kitérőt, mielőtt ismertetnénk a rutinokat, és nézzük meg, mi szükséges ahhoz, hogy egy program *gyors* legyen.

Akármilyen programot írunk, az végső soron mindig egy főprogramból és további alprogramokból, függvények és eljárások gyűjteményéből áll. A programban általában minden benne szereplő alprogram lefut legalább egyszer, de általában többször is. Minden egyes alprogramnak van valamilyen futási ideje, amely a gép sebességétől és a programozás ésszerűségétől függően adott. Ha mármost azt akarjuk, hogy a program gyorsuljon, akkor meg kell próbálni felgyorsítani azokat a részeket, amelyek valószínűleg gyakrabban fognak lefutni. Ha például egy programban van egy **A** és egy **B** eljárás, és az **A** futásának valószínűsége 0,9, míg a **B** futásának valószínűsége 0,1, akkor nyilvánvaló, hogy az **A**-t kell megpróbálnunk minél gyorsabbra megírni, hiszen ez valószínűleg jóval gyakrabban kerül végrehajtásra. A GrPlus egységben is ezt tettük. Kiválogattuk a grafikában gyakrabban használt részeket, és a Graph egység kihasználásával olyan rutinokat írtunk, amelyek lehetőleg gyorsabbak, mint az eredeti. Így a GrPlus egységet a Graph egységgel együtt lehet használni. Ha például egy vízszintes vonalat akarunk húzni, akkor nem érdemes a Line eljárást használni, mert az nem optimalizált erre, míg a Hline kifejezetten erre készült, ezért jóval gyorsabb.

7.3.1. Változók és állandók

```
Unit Grplus;
{ $G+}
{ $L char8x14.obj}
```

INTERFACE

const

```
Lsize = 80;           { 640 pixeles vízszintes felbontást feltételez }

Ink      : byte = 15; { Tintasín a rajzoláshoz }
```

Az *Ink* inicializált változó határozza meg a rajzolás színét. A *setcolor(xx)* eljárás helyett egy egyszerű *Ink:=xx*; utasítással lehet beállítani az írás színét.

```
Paper : byte = 00; { Papírszín a karakterek kiírásához }
```

A *Paper* a karakterkiírásnál kap szerepet. Az egység csak a karakterhatárra történő kiíráskor veszi figyelembe.


```
Over : byte = 00; { Kiíráskor a háttér-/papírszín kiválasztása }
```

Az *Over* határozza meg, hogy a karakterhatárra történő kiíráskor az eljárás megtartsa-e a hátteret, vagy oda a *paper* szín kerüljön.

```
Light : byte = 15; { A keretezett BAR-ok bal és felső oldala }
Back : byte = 07; { A keretezett BAR-ok alapszíne }
Dark : byte = 08; { A keretezett BAR-ok jobb és alsó oldala }
```

Ez a három változó a térhatású vagy a keretezett téglalapok esetén határozza meg a fényes oldal (*Light*), a háttér (*Back*) és a sötét oldal (*Dark*) színét, ahogyan a Windowsban a nyomógomboknál.

```
{ Bitmaszkok }
Bit : Array[ 0..7] of byte=($80,$40,$20,$10,$08,$04,$02,$01);
{ "Első" bájtmaszkok a vízszintes vonalhúzáshoz }
fMsk : Array[ 0..7] of byte=($ff,$7f,$3f,$1f,$0f,$07,$03,$01);
{ "Utolsó" bájtmaszkok a vízszintes vonalhúzáshoz }
lMsk : Array[ 0..7] of byte=($80,$c0,$e0,$f0,$f8,$fc,$fe,$ff);
{ Előre definiált színek a keretezett BAR-okhoz (4 típus) }
Cols : Array[ 0..11] of byte=(15,8,15,8,7,7,3,3,8,15,8,15);
```

A maszktömbök a vonalak és pontok rajzolásakor kapnak szerepet, részletesebben majd ott ismertetjük ezeket.

7.3.2. Eljárások

Az egység a következő eljárásokat tartalmazza:

Procedure WriteMode (Mode:byte);

Az eljárással a VGA írás és olvasás módjait tudjuk beállítani. A *Mode* paraméter a \$3Ce, \$05 Read/Write Mode regiszterbe kerül. Ezenfelül az eljárás a következő beállításokat végzi el:

```
$3C4,$02<=$0F : minden bitplane írásra engedélyezve
$3CE,$08<=$FF : a bitmaszk 255, minden bit engedélyezve
```

Az egység minden rutinja **Mode 2** írás móddal dolgozik. Az eljárások elvégzik az írás mód beállítását, de a visszaállítást már nem. Ezért ha a *graph* egységgel együtt használjuk az egységet, akkor — mivel a *graph* egység a legtöbb esetben **Mode 0** írás

móddal dolgozik — minden *graph*-beli eljárás hívása előtt célszerű kiadni egy *WriteMode(0)* utasítást.

Procedure OutCharXY (X,Y:word;C:char);

Egy 8×14-es karakter kiírása az X,Y pozícióba. Az X karakteres pozíciót jelent, azaz az értékének 0..79 között kell lennie, az Y pixeles pozíciót határoz meg. A kiírás *Ink* színnel történik, ha az *Over*=00, akkor az eljárás a háttérre *Paper* színre állítja, ha *Over*<>00, akkor a háttérre nem módosítja.

Procedure OutStrXY (X,Y:word; S:string);

Egy sztring kiírása adott pozícióba. Az eljárás az X,Y pozíciótól kezdve kiírja az S sztringet a képre 8×14-es karakterkészlettel. Az X karakteres pozíciót jelent (0..79), az Y pedig pixeleset. A kiírás színezése ugyanúgy történik, mint az *OutCharXY* eljárásban.

Procedure _OutCharXY (X,Y:word;C:char);

Egy karakter kiírása pixeles pozícióba. Az eljárás az X,Y pixeles pozícióba kiírja a C karaktert *Ink* színnel, 8×14-es készlettel. A *Paper* és az *Over* értéke érdektelen.

Procedure _OutStrXY (X,Y:word; S:string);

Egy sztring kiírása pixeles pozícióba. Az eljárás az X,Y pozíciótól kezdve 8×14-es karakterkészletet használva *Ink* színnel kiírja az S sztringet. A *Paper* és az *Over* értéke érdektelen.

Procedure Screen_Init;

A képernyő inicializálása, átváltás grafikus módba. Az eljárás bekapcsolja a 640×480/16-os grafikus módot, és **Mode 2** írás módba állítja a VGA kártyát. Ha a *GrPlus* egységet a *Graph* egységgel együtt használjuk, akkor mindkettő inicializálására szükség van, mégpedig a következő módon:

```
i:=Vga;j:=Vgahi;  
InitGraph(i,j,'bgi elérési útvonal');  
Screen_Init;
```

Procedure Vline (X,Y,Long:word);

Függőleges vonal rajzolása. Az X,Y kezdőponttól húz egy *Long* hosszúságú, *Ink* színű, függőleges vonalat. Az X,Y mindig a vonal felső végpontját adja meg, és a *Long* nem lehet nulla.

Procedure Hline (X,Y,Long:word);

Vízszintes vonal rajzolása. Az *X,Y* kezdőpontból húz egy *Long* hosszúságú, *Ink* színű, vízszintes vonalat. Az *X,Y* mindig a vonal bal szélső végpontját adja meg, és a *Long* nem lehet nulla.

Procedure Plot (X,Y:Word;Color:byte);

Egy pixel kigyújtása. Az *X,Y* pozícióba kirajzolja a *Color* színű pontot.

Procedure Box (X1,Y1,X2,Y2:word);

Egy üres téglalap rajzolása. Az *X1,Y1* paraméterek a téglalap bal felső, az *X2,Y2* paraméterek pedig a téglalap jobb alsó sarkának koordinátáit határozzák meg. A rajzolás *Ink* színnel történik.

Procedure SolidBar (X1,Y1,X2,Y2:word);

Egy kitöltött téglalap rajzolása. A paraméterek megegyeznek a *Box* eljárásnál leírtakkal.

Procedure Bbar (X1,Y1,X2,Y2,Ty:word);

Vastagon keretezett, térhatású téglalap rajzolása. A téglalap méretét az *X1,Y1* és *X2,Y2* pozíciók határozzák meg. A színezés a következőképpen történik:

Típus	Bal és felső szél	Jobb és alsó szél	Háttér
0	15, fehér	8, sötétszürke	7, világosszürke
1	8, sötétszürke	15, fehér	7, világosszürke
2	15, fehér	8, sötétszürke	3, sötétcián
3	8, sötétszürke	15, fehér	3, sötétcián
4 vagy több	<i>Light</i>	<i>Dark</i>	<i>Back</i>

Tehát ha a *Ty* paraméter 0..3, akkor az eljárás a *Cols* tömbből veszi az előre definiált értékeket, és ezekkel tölti fel a *Light*, *Dark* és *Back* változókat, ha pedig a *Ty* nagyobb, mint három, akkor az eljárás a változók utolsó értékeit veszi. A keret a megadott téglalap koordinátáin belül van, és két pixel széles. Ha a koordináták helytelenek, akkor a rutin hibásan fog működni, ezért az *X1,Y1* koordinátával mindig a bal felső, az *X2,Y2* párossal pedig mindig a jobb alsó sarok pozícióját kell megadni!

Procedure Cbar (X1,Y1,X2,Y2,Ty:word);

Vékonyan keretezett, térhatású téglalap rajzolása. A színezés és a koordináták megadása ugyanúgy történik, mint a *Bbar* eljárásnál. A keret a megadott koordinátákon belül van, és egy pixel széles.

Function Point (X,Y:Word):byte;

Egy pixel színének lekérdezése. A rutin megfelel a *Graph.Getpixel* eljárásnak.

Procedure ClearScreen;

A grafikus kép törlése. Az eljárás törli a 640×480-as grafikus képet (38 400 színes bájt kinullázása).

7.3.3. Az eljárások működése

Procedure Charset; **External**;

A karakter- és a sztringkiíró eljárásokhoz a kódszegmensben található egy 8×14-es készlet, amelyet a Pascal BINOBJ segédprogramjával alakítottunk át object fájlá, és a \$L direktívával szerkesztettünk hozzá a programhoz. A készletet *Charset* névvel érhetjük el, mind a 256 karakter megtalálható benne, és a megfelelő magyar ékezetes betűket is tartalmazza (CWI kódok).

7.3.4. Az írás mód átváltása

Procedure WriteMode (Mode:word); **Assembler**;

Asm

```

mov    dx,$3c4          { Plane írásmaszk regiszter          }
mov    ax,$0f02
out    dx,ax            { Plane maszk: 00001111b            }
mov    al,5             { Read/Write mode regiszter          }
mov    ah,byte [ Mode]  { A Mode csak 0,1,2,3 lehet!!        }
mov    dx,$3ce
out    dx,ax            { Write Mode                          }
mov    ax,$ff08
out    dx,ax            { Bitmaszk : 11111111b              }

```

End;

A *WriteMode* rutint a megfelelő írás mód beállítására használjuk. Egyébként az összes eljárás **Mode 2** írás móddal dolgozik.

7.3.5. Karakterek kiírása bájtárra

A következő eljárás egy karaktert ír ki karakterpozícióra.

Procedure Char8x14; **Assembler;**

Asm

```
cld
mov    bh,96h
mov    bl,paper          { BX a papírtáblázatra mutat          }
mov    di,sadr           { DI-be rakja a képcímet              }
```

Az *Sadr* belső változó adja a pozíciót. Minden karakter kiírásakor automatikusan növekszik.

```
inc    sadr              { A képcím növelése az esetleges
                        következő íráshoz                      }
mov    ah,14             { 14*kód+k.cím = karakter k.cím      }
mul    ah
lea    si,Charset        { A készlet offsetje az egység
                        kódszegmensében                        }
add    si,ax             { Kiszámítja a karakter címét (SI)    }
```

A CS:SI pointer a karakterkészlet megfelelő címére fog mutatni, a megfelelő karakter bittérképet innen kell kimásolni a képre.

```
mov    dx,03ceh          { A $3CE,$08 bitmaszk regisztert
                        használjuk                              }
mov    al,08h
out    dx,al
inc    dx                { DX a bitmaszk regiszterre mutat    }
```

A DX a bitmaszk adatregisztert címzi meg. Most az *Over* értékéből meghatározza, hogy kell-e a *Paper* szín, vagy meghagyja a háttér:

```
mov    cx,14             { A karakterek 14 pixel magasak      }
mov    al,Ink            { Ez lesz a rajzolás színe          }
cmp    over,00           { Az OVER meghatározza a kiírási
                        módot                                  }
jz     @paper
```

Az *Over<>00* esetben változatlanul hagyja a hátteret.

```
@nopaper:          { OVER<>0, ekkor nincs papírszín      }
    segcs outsb      { A karakter bitminta a bitmaszkba
                      kerül                                }

    mov  ah,ES:[ di]  { Feltölti a latch-et              }
    stosb             { Kiírás a képre                    }
    add  di,Lsize-1   { A képcím növelése                 }
    loop @nopaper     { Kiírja a 14 karaktersort          }
    jmp  @exit        { Kiugrás, ha már kiírta a karaktert }
```

Az *Over=00* esetben *Paper* színre állítja a hátteret.

```
@paper:            { OVER=0, ekkor van papírszín          }
    segcs outsb      { A karakter bitminta a bitmaszkba
                      kerül                                }

    mov  ah,ES:[ bx]  { A latch-et a papírszínnel tölti fel }
    stosb             { Kiírás a képre                    }
    add  di,Lsize-1   { A képcím növelése                 }
    loop @paper       { Kiírja a 14 karaktersort          }

@exit:
    End;
```

Amennyiben az *Over=0*, a karaktert teljesen felülírja a képen. Ezzel tulajdonképpen kiküszöbölhetjük a *Graph.OuttextXY* eljárásnak azt a kellemetlen hatását, hogy a hátteret mindig változatlanul hagyja. Bizonyára más is tapasztalta már, hogy ha az *OuttextXY*-nal írja ki a képre például egy számláló értékét folyamatosan, akkor minden írás előtt törölni kell a kiírási pozíciót, mert különben az előzőleg kiírt érték látható marad. A törlés viszont villogást eredményez. Az *OutCharXY* és az *OutStrXY* eljárással csak karakterhatárookra lehet írni ugyan, de nincs szükség törlésre, mert ha az *Over=0*, akkor a felülírás automatikus. Az eljárás a megadott *Paper* színt viszi ki az *Over=0* esetben. A színeket a *Screen_Init* eljárás állítja elő a grafikus mód bekapcsolásakor, mégpedig úgy, hogy a képernyő utáni területen található 16 színes bájtot beszínezi a lehetséges színekkel. Mivel a képernyő mérete 38 400 bájt, ezért a paper színek a képmemóriában a 38 400-adik bájtól kezdve 16 bájt területet használnak fel. A 38 400 hexadecimálisan 9600h, a program elején ezért töltünk 96h-t a BH regiszterbe és a *Paper* színt a BL regiszterbe. Így a BX mutatja a paper értékéhez tartozó színes bájtot, amelyet a latch-be kell tölteni, ha a hátteret törölni kell, azaz ha az *Over=0*.

Az *OutCharXY* a fenti eljárást hívja meg, miután kiszámolta a koordinátákhoz tartozó címet. Mivel ez a kiírás karakterhatárra történik, ezért az *X* paraméternek bájtthart kell jelölnie (0..79).

Procedure OutCharXY(X,Y:word;C:char); **Assembler;**

Asm

```

mov    ax,Lsize           { Címszámítás a koordináták alapján      }
mul     Y                  { 80*Y                                   }
add     ax,X               { 80*Y+X , tehát az AX a képcímet adja   }
mov     Sadr,ax            { A kiszámított címet az Sadr változóba   }
                           teszi                                     }
mov     al,C               { Az AL-be kerül a karakter kódja       }
mov     ES,SegA000         { Az ES-t a képre állítjuk         }
call    Char8x14           { Kiírja a karaktert                     }

```

End;

Egy sztring kiírása is a *Char8x14* rutinnal történik. Az *OutStrXY* először kiszámítja a kezdőpozíciót és az *Sadr* változóba teszi azt, majd a sztring kiírása következik karakterenként.

Procedure OutStrXY(X,Y:word; S:string); **Assembler;**

Asm

```

cld
mov     ax,Lize             { Címszámítás                             }
mul     Y
add     ax,X
mov     sadr,ax             { A kiszámított cím az Sadr változóba   }
                           kerül                                     }
mov     ES,SegA000
mov     bx,DS               { BX az eredeti adatszegmens címét     }
                           tárolja                                 }
lds     si,S

```

A DS:SI pointer a szövegre mutat. Az első elem a szöveg hosszát adja meg.

```

mov     dx,DS               { DX a sztring szegmenscímét tárolja   }
xor     ch,ch               { CH = 00                               }
lodsb                                { Az első adat beolvasása       }
mov     cl,al               { CX-be kerül a sztring hossza         }
jcxz    @exit               { Kilép, ha a sztring nulla hosszúságú }

```

A következő ciklus sorban veszi a karaktereket, és a *Char8x14* rutinnal kiírja azokat a képre:

```

@outchars:
    mov     DS,dx          { A sztring szegmensét teszi a DS-be      }
    lodsb   { Veszi a következő karaktert                          }
    mov     DS,bx          { Visszatölti az eredeti DS-t              }
    pusha   { Tárolja az általános regisztereket                     }
    call    Char8x14        { Kiírja a karaktert                      }
    popa    { Visszatölti a regisztereket                             }
    loop    @outchars       { Minden karaktert kiír                  }

@exit:
    mov     DS,bx          { A DS-be az eredeti értékét tölti       }
    End;

```

7.3.6. Karakterek kiírása pixelpozícióba

A *_Char8x14* rutin a pixelhatárra történő karakterkiírást valósítja meg. A kiírás kép-címét most is az *Sadr* változó adja, a bájtön belüli pixeles eltolást pedig az *s_x*. A rutin önmagában nem működőképes, a későbbi *_OutCharXY* és *_OutStrXY* fogja hívni egy-egy karakter kiírására. Az előbbi *Char8x14* rutinhoz képest az az alapvető eltérés, hogy a pixelhatár miatt a karaktermintát el kell tolni a megfelelő bitpozícióba.

Procedure _Char8x14; **Assembler;**

Asm

```

    push    DS
    pusha   { Menti a regisztereket                                  }
    mov     ah,14          { A karakter kezdőcímének kiszámítása     }
    mul     ah              { 14*kód                                  }
    lea     si,Charset      { SI= 14*kód+készletcím                  }
    add     si,ax           { Kiszámítja a karakter kezdőcímét      }
    mov     dx,Seg @data    { Az ES-t az adatszegmensre állítjuk    }
    mov     ES,dx          { Az ES-t az adatszegmensre állítjuk    }
    cld
    lea     di,Rbuff        { ES:DI a forgatóbufferre mutat        }
    { Az adatot a pixeles eltolás miatt a forgatóbufferbe másoljuk }
    mov     cx,14           { 14 bájt adatmásolás                  }
    xor     al,al           { Az alsó bájtba 00 kell                }
@tobuff:
    stosb   { A forgatás miatt egy 0 bájt                          }
    segcs   movsb           { A karakter egy sora                  }
    loop    @tobuff         { Kimásolja a készletből a karaktert    }
    mov     al,s_x          { Az S_X alapján a forgatás száma      }
                                számítható
    and     al,7
    mov     cl,8
    sub     cl,al           { CL megadja a balra forgatások számát }
    lea     si,Rbuff        { SI a forgatóbufferre mutat          }

```


{ Az egész buffert elforgatjuk, így a karakterminta pixeles pozícióba kerül}

```

rol    word [ si+00],cl
rol    word [ si+02],cl
rol    word [ si+04],cl
rol    word [ si+06],cl
rol    word [ si+08],cl
rol    word [ si+10],cl
rol    word [ si+12],cl
rol    word [ si+14],cl
rol    word [ si+16],cl
rol    word [ si+18],cl
rol    word [ si+20],cl
rol    word [ si+22],cl
rol    word [ si+24],cl
rol    word [ si+26],cl { A karakter minden sorát elforgatja }
mov    di,Sadr          { DI a képcímet tartalmazza }
inc    Sadr
mov    dx,ES
mov    DS,dx            { DS:SI most már a bufferre mutat
                        (forrás) }

mov    ES,SegA000       { ES:DI most már a képre mutat (cél) }
mov    dx,03ceh         { A bitmaszk regiszter I/O címe a DX-be }
mov    al,08h           { A bitmaszk regiszter a 8-as
}

out    dx,al            { A regiszter címe a $3CE portra kerül }
inc    dx               { A DX a bitmaszk regiszterre mutat,
                        $3CF }

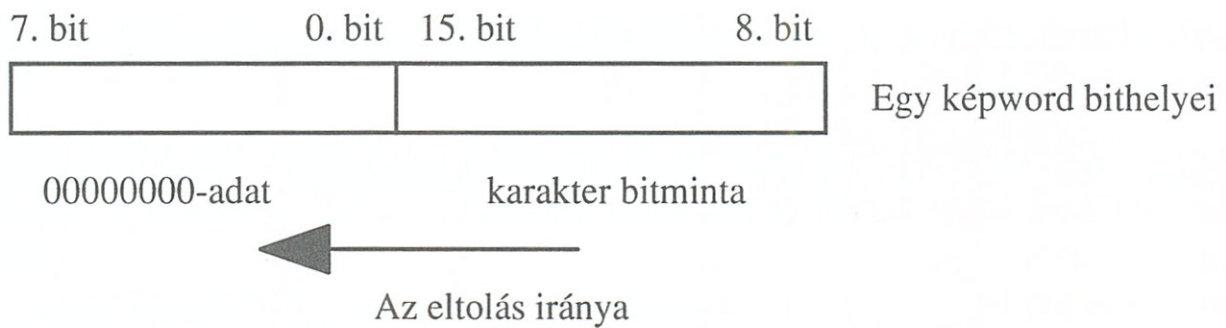
mov    cx,14            { A karakter 14 pixel magas }
mov    al,Ink           { Ez a rajzolás színe }

@outchr:
outsb   { A bitminta a bitmaszkba kerül }
mov     ah,ES:[ di]     { Feltölti a latch-et }
stosb   { Kiviszi az adatot (az alsó bájt) }
outsb   { Felső bájt maszk }
mov     ah,ES:[ di]     { latch feltöltés }
stosb   { A felső bájt kivitele a képre }
add     di,Lsize-2      { A képcím növelése }
loop    @outchr         { A karakter minden sorát kiviszi }
popa
pop     DS              { Visszatölti a regisztereket,
                        és visszatér }

```

End;

A fenti eljárásban látható, hogy a karaktert a kiírás előtt el kell forgatni. A forgatást az *Rbuff* átmeneti adatterületen végezzük. Az eltolás logikája a következő:



31. ábra A karakter beforgatása a pixelpozícióba

A karakter bitmintát a felső bájtban helyezzük el, és a ROL utasítással forgatjuk a megfelelő helyre. A forgatások száma a következő képlettel számítható:

$$\text{Forgatások száma} = 8 - (X \text{ pozíció AND } 7)$$

(Ez a fajta forgatás sajnos csak a 8 pixel széles ábrák esetében használható, később a BOB-programozásnál látni fogjuk, hogy másféleképpen is lehet forgatni.) A forgatás után a karaktert kivisszük a képre, de a *Paper* szín itt már nem számít. Mivel a karakter ilyenkor két bájt széles, ezért 16 bájtot kell kivinni a képre.

Az `_OutCharXY` a fenti eljárást felhasználva ír a képre. Az eljárás elvégzi a képcím (*Sadr*) és a pixeles eltolás (*s_x*) értékének a meghatározását, majd a karaktert kiírja a `_Char8x14` rutinnal.

Procedure `_OutCharXY(X,Y:word;C:char); Assembler;`

Asm

```

mov    ax,Lsize           { Címszámítás a koordinátákból           }
mul     Y                 { AX = 80*Y                               }
mov     di,X              { DI = X koordináta                       }
shr     di,3              { X shr 3 (bájtpozíció)                   }
add     ax,di             { Az AX-ben megjelenik a képcím         }
mov     sadr,ax           { A címet az Sadr változóba tesszük      }
mov     al,byte [ X]      { Az X alsó három bitje a pixeles      }
                                eltolás                               }

and     al,7              { Csak az alsó három bit számít (0..7)   }
mov     s_x,al            { A pixeles eltolást az S_X-be tesszük   }
mov     al,C              { A karakter kódja az AL-be             }
call    _Char8x14         { Kiírja a karaktert                     }

```

End;

Az `_OutStrXY` működése hasonlít az `OutStrXY` működéséhez, de itt természetesen a pixelhatárra kiíró `_Char8x14` rutint használjuk. A címszámítás annyiban különbözik, hogy meg kell határozni a pixeles eltolás mértékét is. Mivel a karakterek 8 bit széle-

sek, minden karakterhez ugyanaz a biteltolás tartozik, ezért ezt elég a legelső karakter esetében meghatározni.

Procedure _OutStrXY(X,Y:word; S:string); **Assembler;**

Asm

```

    cld
    mov     ax,Lsize           { Címszámítás, lásd az _OutCharXY-nál      }
    mul     Y
    mov     di,X
    shr     di,3
    add     ax,di              { A képcím az AX-ben képződik          }
    mov     sadr,ax            { A cím az Sadr-be kerül                }
    mov     al,byte [ X]       { A pixeles eltolás meghatározása        }
    and     al,7
    mov     s_x,al             { A pixeles eltolás az S_X-be kerül        }
    mov     bx,DS              { Az eredeti DS-t a BX-ben tároljuk        }
    lds     si,S               { DS:SI a sztringre mutat                }
    mov     dx,DS              { A sztring szegmensét a DX-ben          }
                                { tároljuk                                }

    xor     ch,ch
    lodsb                                { Az első adat a sztring hossza    }
    mov     cl,al
    jcxz    @exit              { Kilép, ha a sztring 0 hosszúságú        }

@outchars:
    mov     DS,dx
    lodsb                                { Veszi a következő karaktert      }
    mov     DS,bx
    call    _Char8x14          { Kiírja a karaktert                }
    loop    @outchars          { A ciklus minden karaktert kiír        }

@exit:
    mov     DS,bx              { A DS eredeti értéke a BX-ből          }

End;

```

A kétféle karakterkiírás a *Graph.OutText* eljárásaival együtt jó néhány kiírási lehetőséget ad a programozónak. Mindig a konkrét feladat dönti el, hogy melyik fajta kiírást célszerű alkalmazni. Ha az írás méretének a változtatása a cél, és több készlet használatát kell lehetővé tenni, akkor az *OutText* eljárások jól alkalmazhatók, míg az *OutStrXY* és az *_OutStrXY* csak egyetlen készletet használ, de gyors kiírásokra és felülírásra is alkalmas.

7.3.7. A képernyő inicializálása

A *Screen_Init* rutin bekapcsolja a képernyőt. A rutin átvált a 640×480/16-os grafikus módba, és a kép nem látható részében előállítja a *Paper* színekhez tartozó színes bájtokat.

Procedure Screen_Init; **Assembler;**

Asm

```

mov    ax,0012h
int     10h           { 640x480/16-os videomód beállítása a BIOS-szal }
mov     ES,SegA000
mov     di,38400       { A kép mögötti első bájt címe }
cld
mov     dx,03ceh
mov     ax,0205h       { Mode 2-es írás mód beállítása }
out     dx,ax
mov     ax,0ff08h
out     dx,ax          { A bitmaszk $FF-be állítása }
mov     cx,16          { 16 bájtot kell beszínezni }
xor     al,al          { Az AL adja a színt }
@cikl:
stosb           { 16 színes bájt létrehozása: a papírszínek }
inc      al
loop    @cikl
End;
```

7.3.8. Különleges vonalrajzoló eljárások

A *Vline* gyorsított függőlegesvonal-rajzoló eljárás. A gyorsításra a képernyőnek azt a tulajdonságát tudjuk kihasználni, hogy függőleges vonal esetében az egymás alatt lévő képbájtok ugyanazon bitjét kell beállítani.

Procedure Vline(X,Y,Long:word);**Assembler;**

Asm

```

mov     cx,Long
jcxz    @exit          { Kilép, ha a hosszúság nulla }
```

Kiszámítjuk a kezdőcímet a koordináták alapján.

```

mov     ax,Lsize       { Képcímszámítás }
mul     Y
mov     di,X
mov     si,di           { Az X-et az SI-ben is tároljuk }
                        { későbbre }
shr     di,3
add     di,ax           { A DI-ben a képcím található }
```


Meghatározzuk a bitmaszkt is. Ez természetesen mindvégig ugyanaz marad, hiszen a vonal függőleges.

```

and    si,7                { Az X alsó három bitjét tartjuk csak
                             meg }
add    si,offset Bit      { A táblázat kezdőcímét hozzáadjuk az
                             SI-hez }
mov     ah,[ si]           { Az AH-ba beolvassuk a megfelelő
                             bitmaszkt }
mov     al,8
mov     dx,03ceh          { $3Ce,$08: bitmaszk regiszter }
out     dx,ax             { A bitpozíció a bitmaszk regiszterbe
                             kerül }
mov     al,Ink            { A rajzolás Ink színnel fog
                             történni }
mov     ES,SegA000        { Az ES:DI a vonal legelső bájtjára
                             mutat }

```

A bitmaszk regiszter módosítása után nincs más dolgunk, mint az *Ink* színnel *Long* hosszúságban módosítani az egymás alatt lévő képbájtok megfelelő bitjét.

```

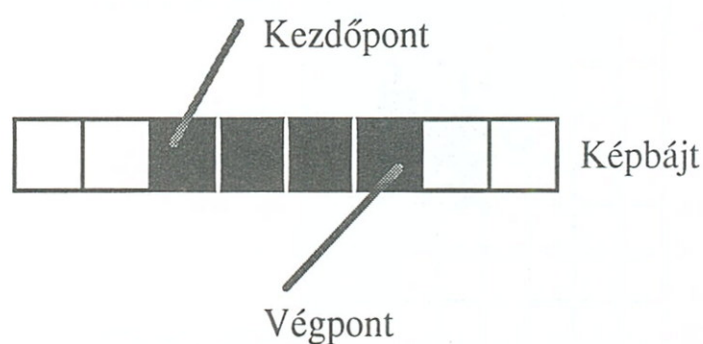
@cikl:
mov     ah,ES:[ di]        { A latch feltöltése }
mov     ES:[ di],al        { A vonal egy pixeljének kiírása }
add     di,Lsize           { A címet növeli }
loop    @cikl              { Kirakja az összes pixelt }
@exit:
End;

```

A következő eljárás, a *Hline* vízszintes vonalat húz, mégpedig jóval gyorsabban, mint a *Graph.Line*. A rutin működése kissé talán bonyolultnak fog tűnni, de az eljárás *tagadhatatlanul* gyors. A vízszintes vonalak rajzolása esetén ugyanis ki lehet használni a képszerkezet azon tulajdonságát, hogy itt bájtok helyezkednek el egymás mellett, és így nem kell pixelenként rajzolni. Ha ugyanis ismerjük a vonal két végpontját, akkor meg lehet határozni az általa elfoglalt képbájtokat!

Egy vízszintes vonal hosszától függően a képmemória szempontjából a következő esetek állhatnak elő:

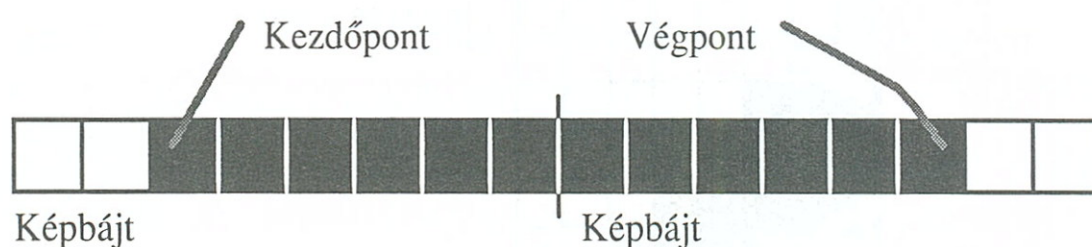
1. A kezdőpont és a végpont ugyanabban a képbájtban helyezkedik el:



32. ábra A vonal elhelyezkedése – egy bájt

Ebben az esetben a képre csupán egyetlen bájtot kell kivinni.

2. A kezdőpont és a végpont két egymást követő bájtban helyezkedik el:



33. ábra A vonal elhelyezkedése – két bájt

Ebben az esetben két egymást követő bájtot kell átírni.

3. A kezdőpont és a végpont bájtjai között n darab 255-ös bájt helyezkedik el.

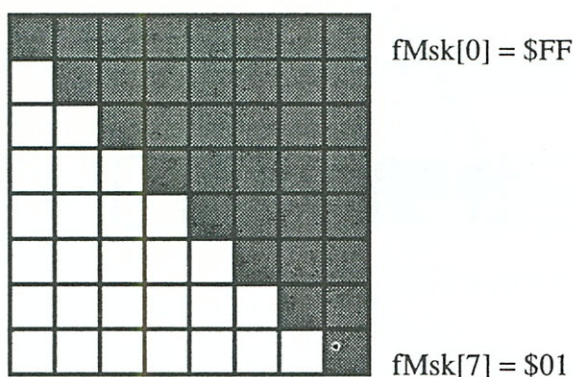


34. ábra A vonal elhelyezkedése – több, mint két bájt

Ebben az esetben $n+2$ bájt módosítására van szükség.

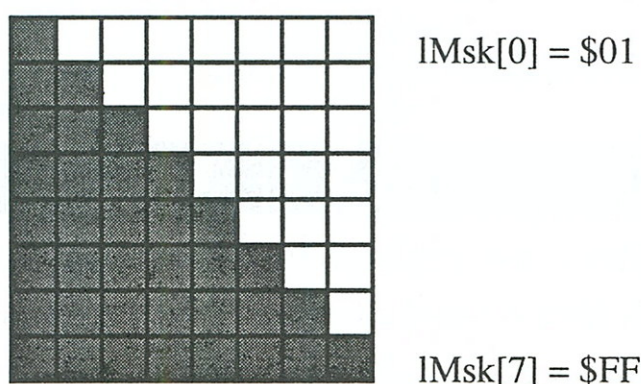
A *Graph.Line* eljárás pixelenként rajzolja a vonalat, míg a *Hline* ezt a bájt kombinációs algoritmust fogja alkalmazni, ezért lesz gyorsabb.

Az első bájtához tartozó értéket egyszerű indexeléssel számítjuk ki az *fMsk* (first mask) tömb segítségével. A tömb a következő értékeket tartalmazza:



35. ábra Az első bájtmaszkok

Az *fMsk* tömbből tehát az *X* and 7 index segítségével ki lehet venni a megfelelő *első bájtmasz*kot. Az *lMsk* tömb viszont az utolsó bájtmaszkokat adja:



36. ábra Az utolsó bájtmaszkok

Ezzel a két tömbbel tehát könnyen elvégezhető a két szélső bájt meghatározása. Hogy a fenti három esetből melyik áll fenn, azt az *X* és az *X+Long* relációja határozza meg:

$((X+Long) \text{ shr } 3) - (X \text{ shr } 3) = 0$ 1. eset

$((X+Long) \text{ shr } 3) - (X \text{ shr } 3) = 1$ 2. eset

$((X+Long) \text{ shr } 3) - (X \text{ shr } 3) > 1$ 3. eset

A képlet egyszerűen meghatározza, hogy a kezdő- és a végpont egymástól hány bájt „távolságra” van a képen. Az első esetben a két maszk AND kapcsolata megadja azt a bitmaszket, amelyet ki kell vinni a képre, a második esetben a két maszket a két egymást követő képbájtra kell kivinni, míg a harmadik esetben 255 értéket kell kiírni a köztes bájtokba.

Procedure Hline(*X,Y,Long:word*); **Assembler;**

Asm

cmp Long, 0

jz @exit

mov ES, SegA000

cld

{ Kilép, ha a hosszúság nulla

{ Az ES mutat a képernyőre

}

}

```

{ Címszámítás }
mov ax,Lsize { Címszámítás a koordináták alapján }
mul Y
mov di,X { DI = X }
mov dx,di { DX = X ez a vonal kezdőpontja }
           pixelesen }
shr di,3 { Az X alsó három bitjére nincs szükség }
add di,ax { DI = képcím (a bal kezdőpont címe) }
mov cx,dx { A CX-be tesszük a kezdőpont X-et }
add cx,Long { Hozzáadjuk a vonal pixeles hosszát }
dec cx { -1, így most a CX a végpont X }
           koordinátája }
{ Az utolsó bájt maszkjának megállapítása }
lea bx,lMsk { Az utolsó bájtmaszk táblázat }
mov al,cl { A végpont X koordinátájának alsó }
           bájtja }
and al,7 { Csak az alsó három bit érdekes }
xlat { A táblázatból kivesszük a maszkot }
mov si,ax { lo(SI) = last mask }
{ Az első bájt maszkjának megállapítása }
lea bx,fMsk { Az első bájtmaszk táblázat }
mov al,dl { A kezdőpont X koordinátájának alsó }
           bájtja }
and al,7 { Csak az alsó három bit érdekes }
xlat { AL = first mask }
{ Most meghatározzuk, hogy a fenti esetekből melyik áll fenn }
shr cx,3 { A végpont bájt pozíciója }
shr dx,3 { A kezdőpont bájt pozíciója }
sub cx,dx { A vonal típusának meghatározása }
mov dx,03ceh { A DX = a bitmaszk regiszter I/O címe }
jz @one_byte { Ugrás, ha az első eset áll fenn }
{ A 2. és a 3. esetet együtt kezeljük }
dec cx { A CX megadja a közbenső bájtok számát }
mov ah,al { Első bájt maszk }
mov al,8 { A bitmaszk regiszter száma }
out dx,ax { A first mask kivitele a regiszterbe }
mov al,ES:[di] { A latch feltöltése }
mov al,Ink { A rajzolás színe }
stosb { Az első bájt kirakása a képernyőre }
jcxz @nx { Ugrás, ha nincsenek közbenső bájtok }
mov ax,0ff08h { A közbenső bájtok maszkja $FF }
out dx,ax
mov al,Ink { A rajzolás színe }
rep stosb { Kirakja a közbenső bájtokat a képre }
@nx:
mov ax,si { AZ SI alsó bájtja kell }
mov ah,al { Az utolsó bájt maszkja }
mov al,8

```



```

out    dx,ax
mov    al,ES:[ di]      { A latch feltöltése }
mov    al,Ink           { A rajzolás színe }
stosb                { Az utolsó bájt kirakása a képre }
jmp    @exit           { Kilépés }
{ Ez akkor fut le, ha az 1. eset áll fenn }
@one_byte:           { Egy bájtban van a kezdő- és a végpont }
mov    bx,si
and    al,bl           { A bitminta az AL-be: AND kapcsolat }
mov    ah,al
mov    al,8            { Majd a bitmaszk regiszterbe }
out    dx,ax
mov    al,ES:[ di]      { A latch feltöltése }
mov    al,Ink           { AL-be a rajzolás színe }
stosb                { A vonal kirakása a képre }
@exit:
    End;

```

7.3.9. Pont kirajzolása és lekérdezése

A *Plot(putpixel)* és a *Point(getpixel)* rutin működését már megismertük a fejezet elején. Az egység rutinjai azonosak az ott megadottakkal.

```
Procedure Plot(X,Y:Word;Color:byte);Assembler;
```

```
    Asm
```

```
        ....
```

```
    End;
```

```
Function Point(X,Y:Word):byte;Assembler;
```

```
    Asm
```

```
        ....
```

```
    End;
```

7.3.10. Téglalapok rajzolása

A *Box* eljárás a *Graph.Rectangle* eljáráshoz hasonlóan működik, csak persze gyorsabb. Az eljárás a *Vline* és a *Hline* hívásával rajzolja ki a téglalapot.

```
Procedure Box(X1,Y1,X2,Y2:word); Assembler;
```

```
    Asm
```

```
    push 2
```

```
    call Writemode      { Kettes írás módba kapcsol }
```

```
    mov  ax,X2          { Kiszámítja a vízszintes hosszúságot }
```

```
    sub  ax,X1
```

```

mov    r0,ax                { A vízszintes méretet az R0 tárolja      }
mov    ax,Y2                { Kiszámítja a függőleges hosszúságot    }
sub    ax,Y1
inc    ax
mov    r1,ax                { A magasságot az R1 tárolja            }
{ A méretek meghatározása után a vonalakat meghúzzuk a
  VLINE HLINE-nal }
push   X1
push   Y1
push   r0
call   Hline                { Felső vízszintes                      }
push   X1
push   Y2
push   r0
call   Hline                { Alsó vízszintes                      }
push   X1
push   Y1
push   r1
call   Vline                { Bal függőleges                      }
push   X2
push   Y1
push   r1
call   Vline                { Jobb függőleges                      }
End;

```

A következő eljárás, a *SolidBar* kitöltött téglalapot rajzol. Az eljárás hasonlóan működik, mint az előbb ismertetett *Hline*. Ugyanúgy meghatározza az első és az utolsó maszkot, valamint a közbenső bájtok számát. Az eltérés mindössze annyi, hogy a kiszámított adatokkal nem egy, hanem $Y2-Y1$ darab vízszintes vonalat kell húzni. A rajzolás az alábbi sorrendben történik:

1. a bal szélső kezdőbájt kirajzolása oszloposan,
2. a közbenső téglalap kirajzolása 255-ös bájtokkal, ha szükséges,
3. a jobb szélső utolsó bájt kirajzolása oszloposan.

Tehát az eljárás kezdeti számításai megegyeznek a *Hline* kezdeti számításaival. A maszkok és a közbenső bájtok meghatározása után kezdjük el a rajzolást!

Procedure SolidBar (X1,Y1,X2,Y2:word); **Assembler;**

Asm

```

push   2
call   Writemode            { Kettes írás módot használunk        }
mov    ES,SegA000           { Az ES-t a képmemóriára állítjuk    }
cld
{ Címszámítás }
mov    ax,Lsize             { Kiszámítjuk az X1,Y1 sarok képcímét }

```



```

mul    Y1                { 80*Y1 }
mov    di,X1             { DI = X1 }
mov    dx,di             { A DX tárolja az X koordinátát }
shr    di,3
add    di,ax             { A kezdőcím a DI regiszterbe kerül }
mov    cx,X2
lea    bx,lMsk           { Az utolsó maszk meghatározása }
mov    al,cl             { Az utolsó pont X koordinátájának
                        alsó bájtja }
and    al,7              { Az alsó három bit számít csak }
xlat                    { A bitmaszk kivétele a táblázatból }
mov    si,ax             { lo(SI) = last mask }
lea    bx,fMsk           { Az első maszk meghatározása }
mov    al,dl             { Az első pont X koordinátájának alsó
                        bájtja }
and    al,7              { Csak három bit kell }
xlat                    { al = first mask }
shr    cx,3
shr    dx,3              { A bájtpozíciók meghatározása }
sub    cx,dx             { A köztes bájtok száma a CX-be kerül }
mov    dx,$3ce
jz     @one_byte         { Ugrás, ha 1 bájtban elfér
                        vízszintesen }

mov    bx,Y2             { Meghatározzuk a téglalap magasságát }
sub    bx,Y1
inc    bx                { BX a függőleges méretet adja }

dec    cx                { CX a közbenső 255-ös bájtok számát
                        adja }
mov    ah,al             { AH-ba az első bájt maszkja }
mov    al,8              { Az AL-be a bitmaszk regiszter száma }
out    dx,ax             { Az első bájtmaszk kivitele }

mov    r0,bx             { Tároljuk a magasságot }
mov    r1,di             { Tároljuk a képcímet }
mov    al,Ink            { A rajzolás színe INK }
{ A következő ciklus kirajzolja a képre a téglalap bal szélét }
@ficikl:
mov    ah,es:[di]        { A latch feltöltése }
mov    es:[di],al        { A képbájt átírása }
add    di,Lsize          { A képcím a következő sorra áll }
dec    bx                { Csökkentjük a függőleges számlálót }
jnz    @ficikl           { Kirajzoljuk az első bájtok oszlopát }
mov    di,r1             { Az eredeti címet visszatöltjük a
                        DI-be }
inc    di                { Eggyel meg kell növelni }

```

```

jcxz  @nx          { Ugrás, ha nincsenek közbenső bájtok }
mov   ax,$ff08     { A köztes bájtok maszkja $FF }
out   dx,ax
mov   al,Ink       { A rajzolás színe INK }

mov   bx,r0        { A magasságot a BX-be töltjük }
mov   r1,di        { A címet tároljuk }
mov   r2,cx        { A szélességet is tároljuk }
mov   r3,di        { A címet az r3-ban is tároljuk }
{ Ez a ciklus a téglalap 255 értékű közbenső bájtjait rakja ki
  a képre }
push  si           { Az utolsó maszkot kitesszük a verembe }
mov   si,cx        { Az SI-be a vízszintes hosszúságot
                   töltjük }
mov   dx,di        { A DX a címet adja }
@mdcikl:
mov   cx,si        { CX = vízszintes hossz }
mov   di,dx        { DI = cím }
rep   stosb        { Egy vízszintes sor feltöltése }
add   dx,Lsize     { A címet növeljük, a következő sorra
                   áll }
dec   bx           { Csökkentjük a függőleges számlálót }
jnz   @mdcikl      { BX darab függőleges sor kell }
mov   dx,$3ce      { A DX ismét a $3CE I/O címet
                   tartalmazza }
pop   si           { Visszahívjuk az utolsó maszkot }
mov   di,r1        { A tárolt cím }
add   di,r2        { A címhez hozzáadjuk a vízszintes
                   hosszúságot }

@nx:
mov   ax,si        { Az AL-be kerül az utolsó bájt maszkja }
mov   ah,al        { Az utolsó bájt maszkja az AH-ba }
mov   al,8         { A maszk kivitele a bitmaszk
                   regiszterbe }

mov   bx,r0        { A BX-be a magasság }
mov   al,Ink       { INK szín }
{ Ez a ciklus a téglalap bal szélső bájtjait rakja ki a képre }
@lacikl:
mov   ah,es:[di]   { A latch feltöltése }
mov   es:[di],al   { A képbájt módosítása }
add   di,Lsize     { A képcím növelése }
dec   bx
jnz   @lacikl      { A jobb szélső bájtok kirakása a képre }
jmp   @exit        { Kilépés }

```



```

    { Rajzolás, ha vízszintesen 1 bájt a szélesség }
@one_byte:
    mov     bx,si
    and     al,bl           { A két maszk AND kapcsolata }
    mov     ah,al           { A maszk az AH-ba }
    mov     al,8            { A bitmaszk regiszter száma 08 }
    out     dx,ax           { Az eredő maszk kivitele }
    mov     cx,Y2           { Meghatározzuk a magasságot (Y2-Y1+1) }
    sub     cx,Y1
    inc     cx              { CX adja a téglalap magasságát }
    mov     al,Ink          { A rajzolás színe INK }
@oncikl:
    mov     ah,es:[ di]     { A latch feltöltése }
    mov     es:[ di],al     { A képbájt módosítása }
    add     di,Lsize        { A képcím növelése }
    loop    @oncikl         { A téglalap kirajzolása }
@exit:
    End;

```

Nos, lehet, hogy az eljárás egy kissé terjedelmes, de ahol csak lehet, érdemes alkalmazni a *Graph.Bar* helyett, mert jóval gyorsabban rajzolja ki a téglalapot.

A *Bbar* egy térszerűen kidomborodó téglalapot rajzol. Az eljárás annyira egyszerű, hogy itt most nem nézzük meg a működését, hiszen csak egyszerű szubrutinhívásokból áll. A fentebb már megismert *SolidBar* rutinnal rajzoljuk fel az alaptéglalapot, a *Vline* és a *HLine* rutinokkal pedig a kereteket.

Nem ismertetjük a *Cbar*-t sem, mert működése szinte azonos a *Bbar*-ével, a különbséget csak a keretek vastagsága jelenti.

```

Procedure Bbar(X1,Y1,X2,Y2,Ty:word); Assembler;

```

```

    Asm

```

```

        ....

```

```

    End;

```

```

Procedure Cbar(X1,Y1,X2,Y2,Ty:word); Assembler;

```

```

    Asm

```

```

        ....

```

```

    End;

```

7.4. Grafikus objektum: BOB

Mi is az a BOB? A legjobban talán úgy lehetne megfogalmazni, hogy egy olyan, $n \times m$ méretű — téglalap alakú — *kétdimenziós* grafikus objektum, amelyet ki lehet rakni a képre, onnan el lehet tüntetni, vagy mozgatni is lehet.

Az elnevezésnek egyébként nincs sok köze az IBM PC-hez, a fejlesztők a BOB szót az Amiga gépektől vették át. Az Amigákban található egy segédprocesszor, amelynek a neve Blitter (Block Image Transmitter), feladata pedig a különleges grafikus objektumok mozgatása a memóriában a DMA ciklusok alatt. A Blitter által mozgatott objektumokat hívják *Blitter Object*-nek, azaz **BOB**-nak. Bár a PC-ben nincs Blitter, az elnevezés mára már annyira általánossá vált, hogy a grafikus objektumainkat mi is nyugodtan nevezhetjük BOB-nak. Egy másik — még régebbi — elnevezés a C64-től örökölt *Sprite* lehetne. Az alapvető különbség a Sprite és a BOB között, hogy a Sprite megjelenítése a hardver feladata, míg a BOB-ot szoftveres úton hozzuk létre!

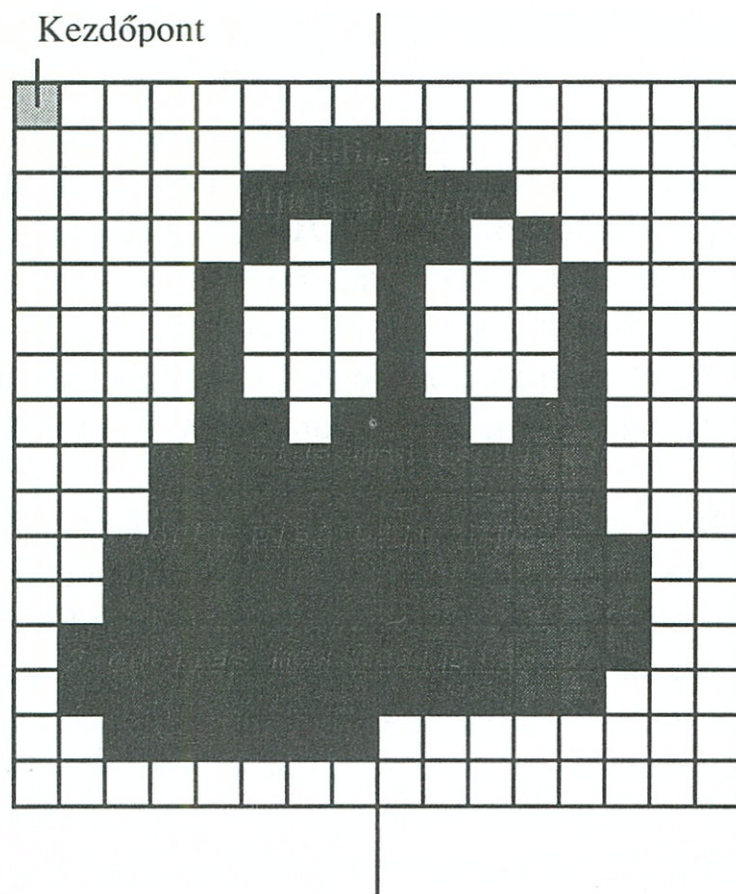
Mind a Sprite, mind a BOB mintáját az ún. Shape (alak, forma) adja meg, amely nem más, mint a BOB grafikus adata a memóriában. Ha tehát megtervezünk egy egyszerű alakot, és eltároljuk a memóriában, akkor az még csak egy Shape — egy alak —, és csak akkor válik BOB-bá, ha ki is rakjuk a képernyőre, és gondoskodunk a mozgatásáról. Egy BOB-nak nevezhető különleges grafikus objektum például az egérkurzor a grafikus módokban. Az egérkurzor egy nyíl alak és egy maszk együttese. Ez a két adattömb az egér-Shape, a megjelenő kurzor pedig a BOB. A mozgatásról, az eltüntetésről és a megjelenítésről különleges rutinok gondoskodnak, ezek a BOB-mozgató és -vezérlő algoritmusok.

7.5. A BOB megjelenítésének nehézségei

Hozzunk létre egy egyszerű, 16×16 pixeles Shape-et! A Shape 16×16 pixeles, ami azt jelenti, hogy vízszintes irányban 16 bitet, azaz két bájtot igényel, és 16 sorból áll, tehát 16×2 bájt memória szükséges az eltárolásához. Mivel a képernyők kezdőpontja általában a bal felső sarok, ezért a BOB-oknak is a bal felső sarkát szoktuk kezdőpozíciónak kijelölni, azaz a BOB X,Y koordinátája ennek a pontnak a koordinátája, és a BOB többi pontja ehhez viszonyul.

A megjelenítést az alábbi részfeladatokra tudjuk bontani:

- A BOB koordinátái alapján meg kell határozni a megfelelő képbájtot, illetve azon belül a megfelelő bitet,
- mivel a Shape-et bájthatárokon tároljuk, ezért a pixelhatáron történő megjelenítés miatt gondoskodni kell a Shape bitenkénti elforgatásáról,
- a megjelenítendő BOB és a képernyő között egyértelmű prioritási sorrendet kell felállítani,
- a megjelenítést általában sebességre kell optimalizálni.



37. ábra Egy egyszerű Shape

Az első részfeladat megoldása viszonylag egyszerű. A pixel megjelenítésének tárgyalásakor már megmutattuk, hogyan kell kiszámolni egy adott X,Y ponthoz a megfelelő képcímet. Alapvetően két megoldás kínálkozik: az egyik az egyszerű szorzás (mint a pixel kigyújtásakor), a másik pedig a táblázattal végzett számítás. A táblázatos számítás gyorsabb, de több memóriát igényel. Mindig a konkrét feladat alapján kell eldöntenünk, hogy melyik megoldást választjuk.

A második lépés szükségessége talán nem mindenki előtt világos. A képmemóriában a bájtok 8-pixeles határokon helyezkednek el. Ha a Shape adatait egyszerűen kimásolnánk a képre, akkor vízszintesen csak 8 pixelenként tudnánk mozgatni a BOB-ot. Ezért van szükség az elforgatásra, vagyis a *megfelelő bithelyre történő eltolásra*. Az egyértelmű, hogy egy 2 bájt szélességű alak elforgatásához szükség van még egy kiegészítő bájtra is, tehát a vízszintes méretet 16-ról 24-re kell megnövelnünk. Ezáltal a Shape memóriaigénye 32 bájtról 48 bájtra nő. Bármekkora is a BOB szélessége bájtokban, az elforgatáshoz ezt mindig meg kell növelnünk egy kiegészítő bájttal. Két megoldás kínálkozik:

- 1: az elforgatást — közvetlenül a BOB kirakása előtt — egy bufferben végezzük el,
- 2: az elforgatást előre elvégezzük, és az elforgatott adatokat tároljuk. Így a BOB megjelenítésénél nem kell időt pocsékolni a forgatásra.

Az első megoldás egyszerű, de nem a leggyorsabb, míg a második megoldás gyors ugyan, de megnyolcszorozza egy Shape memóriaigényét. Ugyanúgy, mint az előbb, itt is a konkrét feladat határozza meg, hogy melyik módszert érdemes alkalmazni.

A harmadik feladat bizony kemény dió. A BOB-ok prioritása akkor fontos, ha ugyanazon a helyen több BOB-ot akarunk megjeleníteni egy időben. Ekkor egyértelműen meg kell adnunk, hogy melyik BOB takarja a másikat. Ez a probléma bonyolódik, ha a BOB alatt a képernyőn valamilyen olyan terület van, amelynek meg kell maradnia. A későbbiekben látni fogjuk, hogy a prioritás a legnagyobb gond az egész BOB-programozás alatt, és minden agyafúrt trükköt fel kell használnunk ahhoz, hogy kielégítően megoldjuk a feladatot.

A negyedik feladat magát a képernyőre történő kirajzolást jelenti. Ezt sokféleképpen meg lehet oldani, de a tapasztalat szerint az egyetlen járható út a bájtonkénti kiírás. A prioritás miatt a kép és a BOB, valamint a BOB és az előtte már kirajzolt BOB-ok között valamilyen logikai kapcsolat van, és ezt kell elvégezni minden egyes BOB bájjal.

7.6. Egyszerű egyszínű megjelenítő

A most következő BOB-megjelenítő a legegyszerűbb megoldás lesz, ennél fogva a „legbutább” is. Nem foglalkozunk a képernyő tartalmával, sem azzal, hogy a BOB-ok takarásba kerülhetnek. A BOB-ok csak egyszínűek lesznek, egyszóval hanyagolunk minden részproblémát, csak primitíven megjelenítjük a BOB-okat. Bár ezt a fajta megjelenítést a gyakorlatban sok mindenre nem használhatjuk, arra azért jó lesz, hogy a járatlan Olvasót bevezesse a BOB-programozásba. (Ezt a típusú megjelenítést akkor tudjuk alkalmazni, ha a BOB-ok biztosan nem ütköznek sem egymással, sem a háttérrel.)

A program elején definiáljuk a BOB-típust, az állandókat és a szükséges változókat:



BOB0.PAS

```
{ $G+}
```

```
Uses crt;
```

Type

```
BOBvar = record           { Egy BOB adatai }
    X,Y:word;              { Koordináták    }
    S,C:byte;              { Minta és szín  }
    dx,
    dy :shortint;          { X és Y elmozdulások }
end;
```


A BOBvar típus a következő adatokat tartalmazza:

X,Y: képkoordináták
S: Shape-szám
C: szín
dX,dY: elmozdulás

Const

```
SegA000 : word = $a000;      { A grafikus kép szegmense }
ScrSize = 80;                { A kép szélessége bájtokban }

BOBnum  = 63;                { Ennyi BOB lesz a képernyőn (+1) }
```

A *SegA000* konstans a grafikus memóriaszegmens címét tartalmazza. A 7.0-s Pascal *System* egysége előre definiálja ezt az állandót, ezért ott nincs rá szükség. Az *ScrSize* a kép logikai szélességét adja meg (640 pixeles felbontásban ez 80), míg a *BOBnum* a BOB-ok száma.

Var

```
i,j      : integer;
f        : file;

Shapes   : Array[ 0..255] of byte;  { A BOB-minták tárolása (8 db) }
Rbuff    : Array[ 0..47] of byte;   { BOB-elforgató buffer }

Bobs     : Array[ 0..BOBnum] of BOBvar; { A BOB-ok adatai }
```

Az *i* és *j* változókat általánosan használjuk. A *Shapes* tömb az alakokat tartalmazza. Ezt a tömböt a lemezen található **bob0.dat** fájl tartalmával töltjük fel, amely 8 db 16×16-os egyszínű Shape-adatot tartalmaz. A *Bobs* tömb a BOB-ok adatait tartalmazza. Az *Rbuff* tömb az átmeneti adattár a BOB-ok elforgatásához.

```
{ A BOB kirakását végző eljárás }
{ Az X,Y pixeles pozícióba kirakja a Shape mintát Color színnel }
```

Procedure SimpleBOB(X,Y:word;Shape,Color:byte); **Assembler;**

Asm

```
{ Az első lépés a Shape átmásolása a forgatóbufferbe }
```

```
mov     ax,DS
mov     ES,ax
lea     si,Shapes      { DS:SI a forráscím lesz }
lea     di,Rbuff       { ES:SI pedig a buffer, azaz a célcím }
mov     bx,di          { A buffer címére később még szükség lesz (BX) }
mov     al,Shape       { A Shape-szám az AL-be }
```

```

xor    ah,ah          { AH=00 }
shl    ax,5           { Szorzás 32-vel }
add    si,ax          { A Shape kezdőcíme: SI = 32*Shape+ SI }
xor    al,al          { AL = 00 }
mov    cx,16          { 16 pixel magas }
cld
    { A Shape-adatot a forgatóbufferbe másoljuk }
@tobuff:
    movsw              { A minta egy sorát másolja át }
    stosb              { A forgatás miatt mögé tesz egy 0 bájtot }
    loop @tobuff       { 16 sort másol át }

    { A második lépés a BOB elforgatása a bufferben a pixeles
      kirakás miatt }

mov    al,byte ptr [X]
and    al,7           { Az AL ( X and 7 ) megadja a forgatások
                      számát }
jz     @norot         { Ha az AL=0, akkor nem kell forgatni }
@rot:
    mov    cx,48       { A bufferben 48 bájt van }
    mov    di,bx       { A DI-t a buffer elejére állítja }
@rotpx:
    rcr    byte ptr [di],1
    inc    di
    loop @rotpx        { Ez a ciklus 1 bittel jobbra tolja a
                      buffert }

    dec    al
    jnz    @rot        { Az eltolást AL-szer kell megismételni }
@norot:
    mov    si,bx       { Az SI most már a forrás lesz (az
                      elforgatott BOB) }

    { A harmadik lépés a képcím kiszámítása és a szín beállítása }

mov    ax,ScrSize
mul    Y
mov    di,X
shr    di,3
add    di,ax          { Kiszámítja a címet: DI=80*Y + X shr 3 }
mov    es,SegA000     { ES:DI a kép megfelelő bájtjára mutat }

mov    dx,$3c4        { A 03c4,02 regiszterrel beállítjuk a
                      színt }

mov    ah,Color
mov    al,02
out    dx,ax          { Write plane enable: a megfelelő
                      bitplane-ek írása }

```



```

    { A negyedik(utolsó) lépés a BOB megjelenítése }
    { Másolás a bufferből a képre }
    mov     cx,16          { Egy BOB 16 pixel magas }
@out:
    movsw
    movsb                { 3 bájt kimásolása a képre: a BOB egy
                        pixelsora }
    add     di,77          { A DI a következő sorra mutat }
    loop    @out           { 16 sort másol ki }
End;

```

Nézzük meg részletesen a *SimpleBOB* eljárás működését! Az eljárás első része a megfelelő Shape-adatot átmásolja a forgatóbufferbe (*Rbuff*). Kiszámítja a Shape kezdőcímét, mivel 1 Shape 32 bájt, így a Shape kezdőcíme **Shapes+32*Shape**, majd úgy másolja át az adatokat (*movsw*), hogy az elforgatás miatt egy 0 bájtot tesz minden Shape-sor után (*stosb*). A második lépés az *Rbuff* megfelelő bithelyre forgatása. A forgatások számát az *X and 7* művelet adja meg: ha az eredmény 0, akkor nem kell forgatni. A *rotpx* ciklus az egész buffer egy pixelrel való elforgatását végzi, míg a *rot* ciklus az összes forgatást adja, tehát lefutása után az *Rbuff* a helyes bithelyre eltolt Shape-adatot tartalmazza. A harmadik lépésben kiszámítjuk a koordinátákhoz tartozó képcímet. Ekkor a DS:SI pointer a forrásadatokra — azaz az *Rbuff*-ra —, míg az ES:DI pointer a célcímre — tehát a képernyőre — mutat. Ezt követően a \$3C4, \$02-es *Plane Write Enable* regiszterrel beállítjuk a megfelelő szint. A negyedik lépésben a buffer tartalmát soronként kimásoljuk a képernyőre. (Az *add di,77* utasítás a DI regisztert az átmásolás után a képernyő következő sorára állítja, mivel a kép logikai hossza 80 bájt, a BOB logikai szélessége pedig 3 bájt.)

```

Procedure MoveBOBS;    { A BOB-okat „egy egyszerű falról 45 fokban }
Begin                  { lepattanó” mozgási algoritmussal mozgatjuk }
    for i:=0 to BOBnum do
        with Bobs[ i] do
            Begin
                inc(x,dx);          { Az X koordináta növelése }
                inc(y,dy);          { Az Y koordináta növelése }
                { Ha a kép szélére ért, akkor irányt kell váltani }
                if (x<3) or (x>620) then dx:=-dx; { X irányváltás }
                if (y<3) or (y>460) then dy:=-dy; { Y irányváltás }
            End;
        End;
    End;

```

A *MoveBOBS* „egy egyszerű falról való lepattanás” mozgatást valósít meg. Minden egyes BOB koordinátáit (X,Y) megnöveljük az elmozdulással (*dX,dY*). Ha bármelyik koordináta a kép széléhez közelít, akkor negálni kell a megfelelő irányú elmozdulást. Az adatgenerálás következtében a *dX* és a *dY* is csak a -1 vagy +1 értéket veheti fel.

```

Procedure ShowBOBS;      { Az összes BOB megjelenítése a képernyőn }
Begin
  for i:=0 to BOBnum do
    with Bobs[ i] do SimpleBOB(x,y,s,c);
  End;

```

A *ShowBOBS* minden egyes BOB-ot megjelenít a saját helyén a saját színével és mintájával.

```

BEGIN
  asm                                { A 640x480/16-os mód beállítása }
    mov  ax,0012h                    { A videomód kódja $12 }
    int  10h                         { A BIOS módbeállító funkció hívása }
  end;

  assign(f,'bob0.dat'); { Beolvassa a BOB-ok mintáit
                           (a Shape-eket) }

  reset(f,1);
  blockread(f,Shapes,sizeof(Shapes)); { Az adatok beolvasása a
                                           fájlból }

  close(f);

```

A főprogram elején beállítjuk a grafikus módot, és betöltjük a **bob0.dat** adatfájlt (inicializáljuk a Shapes tömböt).

```

{ A BOB-ok adatait véletlenszerű adatokkal töltjük fel }
for i:=0 to BOBnum do
  with Bobs[ i] do
    Begin
      x:=3+random(620); { Véletlenszerű X koordináta }
      y:=3+random(440); { Véletlenszerű Y koordináta }
      s:=random(8);      { Véletlenszerű Shape-szám (mintaszám) }
      c:=1+random(15);   { Véletlenszerű szín }
      dx:=(random(2)*2)-1; { X irányválasztás (-1 vagy +1) }
      dy:=(random(2)*2)-1; { Y irányválasztás (-1 vagy +1) }
    End;

```

Inicializáljuk a *BOBS* tömböt, azaz értékeket adunk a BOB-oknak.

```

{ Mozgatási főciklus }
repeat
  MoveBOBS;      { A BOB-ok elmozdítása }
  ShowBOBS;      { A BOB-ok megjelenítése }
  delay(15);     { Késleltetés }
until keypressed;
while keypressed do readkey;

```



```
asm
  mov  ax,0003h      { A 80x25-ös karakteres mód visszaállítása }
  int  10h
end;
END.
```

Ennyi az egyszerű megjelenítő. Az Olvasónak azt javasoljuk, hogy amíg nem érti ennek a programnak minden egyes sorát, addig ne haladjon tovább, mert a továbbiakban ennél bonyolultabb programok következnek.

Vegyük sorra a megjelenítés hátrányait:

- A BOB-ok egyszínűek.
- Felülírjuk a BOB-ok alatti képterületet.
- A BOB-ok találkozása villogással, zavarral jár.
- A BOB-ok elforgatása és a címszámítások sok időt emésztene fel.
- A BOB-ok csak egypixeles elmozdulással mozgathatók, mert különben szemetet hagynak maguk után. Ez a probléma magával hozza azt is, hogy a BOB szélén egy-egy pixel széles háttér színű keretet kell hagyni a törlés miatt. Ez azt jelenti, hogy egy 16×16 pixeles Shape csak egy 14×14-es képet tud tárolni.

A továbbiakban fokozatosan megoldjuk ezeket a feladatokat, ezáltal a megjelenítő egyre bonyolultabb lesz.

7.7. A háttér megőrzése: kettősbuffer-technika

Hogyan lehetne megőrizni a képernyő tartalmát, hogy a BOB-ok úgy mozogjanak a képen, hogy közben az azon lévő adatok ne módosuljanak? A válasz egyszerű: el kell tárolni azt a területet, ahová kitesszük a BOB-ot, és az elmozdításkor ezt a kivágott részt vissza kell tenni a helyére. Ezt még meg is tudnánk oldani, de a villogás szempontjából ez csak olaj lenne a tűzre. Vizsgáljuk meg a mozgatáskor bekövetkező villogást, mint alapjelenséget, és nézzük meg az okait is!

Vegyük egy egyszerű példát! Legyen az a feladat, hogy egyetlen pontot kell mozgatni a képen, mondjuk vízszintesen. A megoldás nem nehéz: a pont utolsó helyére kirakunk egy fekete pixelt, míg az új helyre egy fehéret. Ezt kell ismételtetnünk végig a mozgatás során. Eltüntetés és megjelenítés, vagy megjelenítés és eltüntetés? Ha először rakjuk ki a fekete pixelt és utána a fehéret, akkor a két művelet közötti időben a képen nem látszik a pont. Ennek a sorrendnek az eredménye tehát **villogás**. Ha elő-

ször a fehér pixelt rakjuk ki és csak utána a feketét, akkor viszont egy rövid időre két pont látszik a képen. Ennek a sorrendnek az eredménye tehát **szellemkép**. Hogy melyik lép fel, az végső soron mindegy, mindkettő zavaró hatású. Mi tehát a megoldás? Pontosan szinkronban kell haladni az elektronsugárral, és a változtatásokat akkor kell elvégezni, amikor a sugár biztosan nem jár ott, ahol mi módosítunk. (Emlékezzünk vissza a paletta programozására!)

Ha tehát minden BOB elmozdításakor egyszerűen visszatennénk a képernyő elmentett részét, majd újra kitennénk a BOB-ot, akkor ez villogással járna. Csak egy *nagyon* gyors gép tudná ezt úgy elvégezni, hogy ne legyen villogás.

A villogás és a szellemkép kiküszöbölésére a kettősbuffer- (Double-Buffer) technikát kell alkalmaznunk. Nézzük a lényegét! Létrehozunk két teljesen egyforma képet. Az egyiket nevezzük *aktív*nek, ez az **A** kép, a másikat pedig *inaktív*nek, ez a **B** kép. Az **A** kép éppen látható. Most kirajzoljuk a BOB-okat a **B** képre. A rajzoláskor természetesen ügyelünk a kép és a BOB logikai kapcsolatára, azaz a takarásra. A BOB-ok alatti területet valamilyen formában eltároljuk, hogy az elmozdításakor majd vissza tudjuk állítani a képet az eredeti formájára. Amikor a BOB-ok már kint vannak a képen, akkor megjelenítjük az *inaktív* képet, és logikailag megváltoztatjuk az elnevezését is. Mivel most már ez látszik, ezért a **B** lesz az *aktív*, és az **A** az *inaktív*. Most elvégezhetők az elmozdítások. Az elmozdítások után a módosított helyeken ismét megjelenítjük a BOB-okat az **A** képen, miközben a **B** látszik. Azaz a kép módosításából mi semmit sem látunk, hiszen az a nem látható képen zajlik le. Amikor az **A** kép elkészült, akkor megjelenítjük, ezáltal a BOB-ok elmozdulnak. Ezt a műveletsort végezzük most már folyamatosan. Az *inaktív* képre rajzolunk, az *aktív* látszik. Természetesen mindezt az elektronsugár függőleges visszafutásához, azaz a képváltáshoz kell szinkronizálnunk, máskülönben az egész nem ér semmit. Ez a megjelenítési módszer a logikájából következően garantálja azt, hogy bármennyi BOB is van a képen, és bármilyen sokáig is tart a megjelenítés, sem villogást, sem szellemképet nem fogunk produkálni, legfeljebb a mozgás a túl sok BOB miatt lassú és lépcsőzetes lesz.

Térjünk vissza egy pillanatra a BOB-ok alatti terület eltárolására. Ezt el lehet végezni úgy, hogy a BOB kirajzolásakor elmentjük az alatta lévő területet, de a tapasztalat szerint ez csak kevés BOB esetén ajánlatos, máskülönben a művelet lelassítja az egész megjelenítési folyamatot. Ha a BOB-ok sokan vannak, akkor érdemesebb még egyszer eltárolni az egész képet (legyen ez a **C** kép), azaz a *kettősbuffer-technikát átalakítjuk hármassbuffer-technikára*, és a BOB-ok kirakásakor csak azt a címet jegyezzük meg, ahová a BOB-ot kitettük. A képek helyreállításához tehát a címek alapján a **C** képről vissza kell csak másolnunk az adatokat az *inaktív* képre. Ha a BOB-ok nagyon sokan vannak, akkor még ennél is jobb megoldás, hogy az egész **C**

képet átmásoljuk, így még a címeket sem kell eltárolnunk. Hogy a két megoldás közül melyik a gyorsabb, az csak a BOB-ok számától függ. A tárolásnál és a másolásnál hasznunkra van a VGA kártyának az a tulajdonsága, hogy a képmemóriát képes színes bájtontként másolni, azaz nem kell a másolással bitplane-enként szöszmötölnünk (lásd írás módok: **Mode 1**). Persze ezt csak abban az esetben tehetjük meg, ha az összes képet a VGA kártya számára fenntartott memóriában, azaz az A000:0000-B000:FFFF címeiken tároljuk. A BOB-ok kirakásakor pedig azt a tulajdonságot használjuk ki, hogy a VGA képes egy adatot egyszerre több bitplane-re is felvinni (lásd írás módok: **Mode 2**).

A most következő program a kettősbuffer-technika felhasználásával mozgatja az egy színű BOB-okat. A megjelenítés most már valóban Sprite hatású, mert a BOB-ok a kép „felett” mozognak (mint az egérkurzor), és az egymáson való áthaladásuk is villogásmentes. A BOB-ok prioritását egyszerűen az határozza meg, hogy milyen sorrendben rakjuk ki őket. Amelyik BOB-ot legelőször jelenítjük meg, azt az összes többi takarni fogja, tehát a legelső BOB-nak a legkisebb a prioritása, míg a legutolsó-nak a legnagyobb.



BOB1.PAS

{ \$G+}

Uses Crt;**Type**BOBvar = **record**

```

    X,Y : word;   { [bx+0],[bx+2]; koordináták }
    S,C : byte;   { [bx+4],[bx+5]; mintaszám és szín }
    AdA : word;   { [bx+6] az "A" képhez tartozó utolsó cím }
    AdB : word;   { [bx+8] a "B" képhez tartozó utolsó cím }
    Ir  : byte;   { A mozgás iránya }
    Lp  : byte;   { Az irányhoz tartozó lépésszám }
end;

```

A *BOBvar* típust ki kellett bővíteni néhány új mezővel. A módosított rekord mezői:

X,Y:	koordináták a képen
S,C:	Shape és Color
AdA:	az A képen utoljára kiszámított cím
AdB:	a B képen utoljára kiszámított cím
Ir:	mozgásirány
Lp:	a mozgás irányában megtett lépések száma

Const

BOBsize = SizeOF(BOBvar);

Var

```

i,j,k      : integer;           { Általánosan használt változók }
f          : file;

BOBS       : Array [ 1..BOBnum] of BOBvar; { BOB adatok           }
Shapes     : Array [ 0..1023] of byte;     { Mintaadatok           }
Rbuff      : Array [ 0..47] of byte;      { Forgatóbuffer         }

```

A `_320x240_c16` eljárás beállítja a 320×240/16-os nem szabványos grafikus video-módot.

Procedure `_320x240_c16`; **Assembler**;**Asm**

```

mov     ax,12h           { A 640x480-as módból indulunk ki           }
int     10h              { BIOS módbeállítás                         }
mov     dx,03c4h         { Órajel mód változtatás ( SeqRESET!!)    }
mov     ax,0901h         { 320 pixeles órajel                         }
out     dx,ax
lea     si,_Data_240     { A 320x240-es felbontás CRTC értékei      }
mov     cx,018h          { Ennyi regisztert fogunk átírni (CRTC)    }
mov     dx,03d4h
mov     ax,0011h         { A 7.bit=0, engedélyezi a 0..7          }
                        { regisztereket                               }

out     dx,ax
xor     al,al

@Set_regs:              { A CRTC regiszterek beállítása             }
mov     ah,[ si]         { A táblázat következő eleme             }
out     dx,ax            { A regiszter módosítása                   }
inc     al               { A következő regiszterszám               }
inc     si               { A következő táblázatérték               }
loop    @Set_regs
mov     dx,03ceh         { A Mode 2 írás mód beállítása           }
mov     ax,0205h         { $3CE,$05 Read/Write Mode regiszter      }
out     dx,ax

End;

```

A `Screen_Init` eljárás inicializálja a videomemóriát, valamint feltölti a **B** és a **C** képet.

Procedure `Screen_Init`; **Assembler**;**Asm**

```

{
  Memóriafelosztás:
    "A" kép : A000:0000
    "B" kép : A000:2580   normalizálva : A258:0000
    "C" kép : A000:4B00   normalizálva : A4B0:0000
}

```

```

mov     bx,DS
mov     ES,SegA000      { Mindkét szegmensregiszter a képre mutat
    }
mov     DS,SegA000
mov     dx,03c4h        { Write Plane Enable regiszter          }
mov     ax,0f02h
out     dx,ax           { Valamennyi plane írásra engedélyezve   }
mov     dx,03ceh        { Bitmaszk regiszter                    }
mov     ax,0008h
out     dx,ax           { A maszkbitek törlése (másolás lesz)    }
cld
mov     cx,9600          { Egy kép 9600 színes bájt hosszúságú   }
xor     si,si            { Az "A" kép kezdete                    }
mov     di,9600          { A "B" kép kezdete                    }
rep movsb               { Az "A" kép átmásolása a "B" képre      }
mov     cx,9600
rep movsb               { A  "B" kép átmásolása a "C" képre      }
mov     DS,bx
    { Most mindhárom kép ugyanazt tartalmazza                    }
End;
```

A *Putpixel* egy egyszerű pontkigyújtó eljárás, amelyet a háttérre rajzolás miatt használunk. Az eljárás azonos a **GrPlus** egység leírásánál megismerttel, csak ebben az esetben az *Lsize=40*.

Procedure PutPixel(X,Y:word;color:byte); **Assembler;**

Asm

.. a már megismert pixelrajzoló eljárás..

End;

A program lelke a *ShowBOBS* eljárás. Egyszeri meghívásakor automatikusan kirajzolja a BOB-okat az inaktív képre, majd láthatóvá teszi azokat, azaz képet vált.

Procedure ShowBOBS; **Assembler;**

Asm

```

push    DS
push    bp              { A DS és a BP regiszter tárolása a veremben }
mov     dx,03ceh
mov     ax,0008h
out     dx,ax           { A bitmaszk regiszter nullázása, ekkor
                        másolni fog                                }

mov     bp,DS           { A BP az eredeti adatszegmenset tárolja   }
mov     ES,Inactive     { Az ES az inaktív képre mutat            }
mov     dx,ScrBuff       { A DX a bufferkép szegmensét tárolja    }
```



```

lea    bx,BOBS      { A BX a BOB adatokra mutat }
add    bx,Act        { Ez a korrigáló érték 0/2 (AdA vagy AdB
                      kijelölése) }

cld

{ Az első lépésben a címek alapján helyreállítja az inaktív képet}

mov    ax,BOBnum     { Ennyi BOB van }
@restore:
mov    DS,bp
mov    di,[bx+6]      { Kiveszi a megfelelő címet }
add    bx,BOBsize     { A BX a következő BOB adatra mutat }
mov    si,di          { Az offsetek megegyeznek }
mov    DS,dx          { A DS-t a "C" bufferképre állítja }
mov    cx,16          { Egy BOB 16 pixel magas }
@res:
movsb                     { !! A színes bájtok másolása !! }
movsb                     { 3 darab bájt , azaz 24 pixel }
movsb
add    di,Lsize-3      { Címnövelések }
add    si,Lsize-3
loop  @res             { Egy BOB alatti terület helyreállítása }
dec    ax
jnz    @restore        { Minden BOB-ot letöröl az inaktív képről }

{ A második lépésben feldolgozza a BOB-ok adatait, kirakja a BOB-okat}

mov    DS,bp          { A DS-be visszatölti az eredeti adat-
                      szegmenset }

lea    bx,BOBS
mov    cx,BOBnum       { Ennyi BOB lesz }
@main:
push   cx              { Ez a fő megjelenítési ciklus kezdete }
push   cx              { A BOB-számlálót tároljuk a STACK-ben }
mov    si,Act           { Az AdA/AdB mutató (0/2) az SI-be }
mov    ax,Lsize         { Képcímszámítás }
mul    word[bx+2]       { [bx+2] az Y koordináta }
mov    dx,[bx]          { [bx+0] az X koordináta }
shr    dx,3             { Az alsó 3 bit kiforgatása }
add    dx,ax            { Kiszámolja és a DX-ben tárolja a képcímet }
mov    [bx+si+6],dx     { Beteszi a BOB adatokba is (AdA vagy AdB) }
{ A Shape adatot bevisszük az Rbuff forgatóbufferbe }
lea    di,Rbuff         { Célcím }
lea    si,Shapes        { Forráscímszámítás az S BOB-mező mintaszáma
                      alapján }
mov    al,[bx+4]        { [bx+4] az S mintaszám }
xor    ah,ah            { AH = 00 }
shl    ax,5             { Az S-et megszorozzuk 32-vel }
add    si,ax            { Meghatározza a Shape kezdőcímét:
                      SI regiszter }
xor    al,al            { AL = 00 }

```

```

mov     ES,bp          { Az ES:DI az Rbuff-ra mutat }
{ A Shape bemásolása az Rbuff-ba }
mov     cx,16          { A BOB 16 sorból áll }
@tobuff:
movsw                   { A Shape adatok bemásolása }
stosb                  { Egy kiegészítő nulla adat a forgatáshoz }
loop    @tobuff        { A Shape-et bemásolja a forgatóbufferbe }
mov     ES,Inactive    { Az ES ezt követően az inaktív képre mutat }
mov     al,[bx]         { X and 7 megadja a fogratások számát }
and     al,7           { Az alsó három bit megadja a forgatások számát
                        (0-7) }
jz      @norot         { Ugrás tovább, ha a mintát nem kell
                        elforgatni }
{ Az Rbuff megfelelő bithelyre forgatása }
@rot:
mov     cx,48          { Az Rbuff 48 bájt hosszú }
lea     si,Rbuff       { Az SI-be a kezdőcím }
@rotpx:
rcr     byte [si],1    { Egy pixellel jobbra toljuk minden egyes
                        bájtját }
inc     si             { A buffer következő bájtja }
loop    @rotpx
dec     al
jnz     @rot           { Elforgatja az Rbuff tartalmát }
@norot:
lea     si,Rbuff       { DS:SI a forrásadatokra mutat }
mov     di,dx          { ES:DI pedig az inaktív kép megfelelő
                        bájtjára }
mov     dx,03ceh
mov     al,08h         { Kijelöljük a bitmaszk regisztert }
out     dx,al
inc     dx             { A DX most a bitmaszk regiszter adatot
                        mutatja }
mov     ah,[bx+5]      { Az AH-ba töltjük a színt }

{ A BOB kirakása a képre }
mov     cx,16          { A BOB 16 pixelsor magas }
@putbob:
outsb                   { Az Rbuff-ból a maszkregiszterbe másolunk }
mov     al,ah           { Az AL-be betöltjük a színt }
xchg    al,ES:[di]     { Csere, ez olvasás és írás egyben!!
                        (1. bájt) }
outsb
mov     al,ah
xchg    al,ES:[di+1]   { 2. bájt }
outsb
mov     al,ah
xchg    al,ES:[di+2]   { 3 bájtot rak ki, ennyi a BOB 1 pixelsora }

```



```

add    di,Lsize    { A képcím a következő sorra áll }
loop   @putbob     { 16 pixelsort rak ki }

pop     cx          { Az eredeti BOB-számláló visszatöltése }
add     bx,BOBsize
loop   @main       { A @main ciklus minden BOB-ot kitesz }

{ Most kell megcserélni a két képet, azaz láthatóvá tenni az
  inaktívat }
mov     dx,03d4h    { 3D4,0C,0D: kép kezdőcím regiszter }
mov     al,0ch
mov     ah,byte [ InacScr+1] { Cím HI bájt $3D4, $0C }
out     dx,ax
mov     al,0dh
mov     ah,byte [ InacScr]   { Cím LO bájt $3D4, $0D }
out     dx,ax

mov     dx,03dah    { Várakozás a vertikális visszafutásra }
@wait:
in      al,dx       { A státuszregiszter beolvasása }
and     al,8        { A vertikális visszafutás bit megtartása }
jz      @wait       { Visszaugrás, amíg nincs visszafutás }

{ Cserék }

xor     Inactive,0258h { A szegmenscím cseréje }
xor     InacScr,2580h  { A képkezdőcím cseréje }
xor     Act,2          { A címtárkorrigáló adat cseréje }

pop     bp          { Visszatölti a DS és a BP eredeti
                    értékét }

pop     DS
End;

```

A *MoveBOBS* a BOB-ok mozgatásáról gondoskodik. A mozgatás algoritmus a következő: a 8 lehetséges irányból véletlenszerűen kijelölünk egyet (*Ir*), és meghatározzuk az ebbe az irányba teendő lépések számát (*Lp*). Ha az elmozdulás következtében a BOB falnak ütközik, akkor visszafordítjuk, ha pedig lelépte a lépésszámot, akkor új irányt és lépésszámot adunk neki.

```

Procedure MoveBOBS;    { A BOBok mozgatása }
Begin
  for i:=1 to BOBnum do
    with Bobs[ i] do
      Begin
        { Ütközéskor a BOB meghatározott irányt kap }
        if x<2 then Ir:=2; { A kép bal szélének vizsgálata }

```

```

    if x>302 then Ir:=6; { A kép jobb szélének vizsgálata }
    if y<2 then Ir:=4; { A kép tetejének vizsgálata }
    if y>222 then Ir:=0; { A kép aljának vizsgálata }

    dec(Lp); { Csökkentjük a lépésszámot }
    if Lp=255 then { Ha lejárt a lépés, akkor új adatokat
                    kap }

    Begin
        Lp:=20+random(40); { 20<=Lp<=60 ; új lépésszám }
        Ir:=random(8); { Ir = 0..7 ; új irány }
    End;
    inc(X,dx[ Ir] ); { Elmozdítás }
    inc(Y,dy[ Ir] );
End;
End;

BEGIN
    _320x240_c16; { 320x240/16-os grafikus módba kapcsol }

    assign(f,'bob0.dat'); { Beolvassa a BOB-ok mintáit
                           (a Shape-eket) }

    reset(f,1);
    blockread(f,Shapes,filesize(f));
    close(f);

    { 1000 db véletlenszerű pont kirakása a képre: ez a háttérkép
      lesz }
    for i:=0 to 999 do Putpixel(random(320),random(240),1+random(15));

    Screen_Init; { A képernyők inicializálása ("B" és "C" kép) }

    Randomize; { A BOB-ok adatainak feltöltése }
    for i:=1 to BOBnum do
        with BOBS[ i] do
            Begin
                X:=2+random(300); { Egy véletlenszerű X koordináta }
                Y:=2+random(220); { Egy véletlenszerű Y koordináta }
                S:=random(8); { Mintaszám 0..7 között }
                C:=1+random(15); { Szín 1..15 között }
                Ir:=random(8); { Véletlenszerű irány 0..7 között }
                Lp:=random(40)+10; { Véletlenszerű lépés 10..49 között }
                AdA:=0; { Az "A" kép címe 0000 (csak a
                           kezdéshez) }
                AdB:=0; { A "B" kép címe 0000 (csak a
                           kezdéshez) }
            End;
        end;
    end;
End;

```



```

{ Fő mozgatósi ciklus }
repeat
  MoveBOBS;           { Elmozdítás           }
  ShowBOBS;           { Megjelenítés         }
  delay(10);          { Késleltetés          }
until keypressed;
while keypressed do readkey;

textmode(c80);
END.

```

Tisztázzunk néhány kérdést a programmal kapcsolatban!

1. Miért használtunk 320×240-es felbontást? A kettősbuffer-technika sajnos nagyméretű memóriaterületeket igényel a képek tárolásához. Mivel a képtárolást színes bájtokon oldjuk meg, ezért kénytelenek vagyunk a képmemóriát használni erre a célra. Az természetes, hogy egy kisebb felbontású kép kevesebb memóriát igényel. A 640×480-as mód 38 400 színes bájt memóriát igényel, míg a 320×240-es csak 9600-at. Ez az egyik lényeges szempont. Másrészt a 640×480-as módot nem is lehetne alkalmazni, mert egyetlen kép hossza túllépi a 32 Kbájtot, márpedig az **A** és a **B** képnek mindenképpen az A000 szegmensben, azaz ugyanazon 64 Kbájton belül kell elhelyezkednie. Erre azért van szükség, mert nem minden VGA kártya tudja folyamatosan megjeleníteni a 128 K videoterületet. Ez a tény korlátozza a kettősbuffer alkalmazásának lehetőségeit. A legnagyobb felbontás, amelynél 16 színben alkalmazni tudnánk, 640×400 lenne, de ez szintén nem szabványos mód. Egy lehetséges másik felbontás a szabványos 640×350-es EGA mód, itt viszont a pixeltorzulás okoz gondot. Az alapfogalmaknál már volt szó róla, hogy a négyzet lenne az ideális pixelalak, csak hogy ez nem mindig teljesül. A pixel akkor négyzet alakú, ha a kép és a felbontás vízszintes—függőleges aránya megegyezik. A jelenlegi szabványok szerint a képcsövek 4/3 oldalarányú téglalap alakúak, azaz a vízszintes és a függőleges hossz aránya 1,33. A következő táblázat bemutatja a különböző felbontások arányait:

Szabványos felbontás	Pixeltorzulási hányados
320×200	1.6
640×200	3.2
640×350	1.82
640×480	1.333..
Nem szabványos felbontás	Pixeltorzulási hányados
320×240	1.333..
320×350	0.914
320×400	0.8
320×480	0.666..
640×400	1.6

A táblázat adataiból kiderül, hogy a pixelek csak két felbontásban négyzet alakúak, a 320×240-es és a 640×480-as esetben. Amennyiben a pixeltorzulási hányados kisebb 1.33-nál, akkor a pixel fekvő téglalap alakú, ha pedig nagyobb, akkor álló téglalap alakú.

Mindezeket figyelembe véve a következő érvek szólnak a 320×240-es mód mellett:

- kis memóriaigényű ($65\,536/9600=6.82$, tehát 6 önálló képet lehet létrehozni),
 - négyzetes pixelek alkotják,
 - biztosan beállítható minden kátyán.
2. Hogyan történik a képek váltása? A képeket egyszerűen a $\$3D4, \$0C, \$0D$ képkezdőcím regiszterrel váltogatjuk. Nézzük meg még egyszer a váltás folyamatát! Először beírjuk az új képcímet a regiszterekbe, majd várakozunk az elektronsugárra. A sorrend ellentmondást sugall, hiszen ez azt jelenti, hogy a címet már beírtuk, és a képváltásnak semmi köze a visszafutáshoz. A logikus sorrend az lenne, hogy előbb megvárjuk a visszafutást, majd csak utána írjuk be az új képcímet. Ha a két programrészt megcseréljük, akkor villogás és káosz az eredmény. Miért? A programbeli látszólagos sorrendet a $\$3D4, \$0C, \$0D$ címregiszter természete adja. Ez ugyanis Strobe (impulzusvezérelt) jellegű regiszter, amelynek beírójele a vertikális visszafutási impulzus. Ez azt jelenti, hogy bármikor írunk is a regiszterbe, a kártya csak a visszafutáskor fogja figyelembe venni az új értéket, azaz csak egy új kép kezdésekor lép életbe a beírt új érték. A járatosabb Olvasó bizonyára megérti, hogy ez feleslegessé teszi a visszafutásra való várakozást, hiszen a kép úgyis csak a visszafutáskor fog váltani, és mi éppen ezt akartuk. Ha azonban ezen felbuzdulva kiiktatjuk a várakozást, akkor ismét villogás az eredmény. Az utólagos várakozásra azért van szükség, mert a képnek biztosan váltania kell, mielőtt elkezdenénk rajzolni az új inaktív képet. Ha nincs várakozás, úgy valószínű, hogy akkor fogjuk elkezdeni az inaktív kép újrarajzolását, amikor az még látszik. Ennek az eredménye a villogás, ezért tehát a címbeírás után meg kell várni egy valódi képváltást.
3. Hogyan történik az inaktív kép helyreállítása? A képcímek kiszámításakor az *AdA* és az *AdB* tömbökben tároljuk a címeket. Hogy a kettő közül melyikben, azt az *Act* változó mondja meg, amelynek az értéke 0 vagy 2 lehet. A visszaállításkor visszatöltjük ezt a címet, és a *C* képről visszamásoljuk a színes bájtokat. Noha a képek mind az A000 képszegmensben vannak, a másolás érdekében normalizáljuk a címeket, azaz mindegyik kép 0000 offsettel kezdődik, így a címek minden képen ugyanazok. A visszamásolást három **movsb** utasítás végzi el. Sajnos a videomemóriában nem tudjuk használni a **movsw** utasítást — noha ugyanannyi órajel alatt dupla annyi adatot mozgat —, mert a VGA-latch 8 bites. Ezért minden ilyen másoláskor a **movsb** utasítást kell alkalmazni. Az írás módoknál láthattuk, hogy a képmemória-képmemória adatmozgatásra az egyes mód van fenntartva, de ezt más módoknál is meg lehet oldani, csak nullázni kell a bitmaszk regisztert ($\$3CE, \$08$). Mivel a BOB-ok kirakása kettes módban történik, ezért a másolásra is ebben a módban kerül sor, de a bitmaszk ekkor nulla.

4. Milyen logikai kapcsolat van a BOB-ok és a kép között? A BOB—BOB kapcsolat egyszerűen leírható. A később kirakott BOB-oknak nagyobb a prioritásuk a korábban kirakottakkal szemben. A képet mindegyik takarja, azaz minden BOB a bitplane-ek felett látszik. Ez egyébként a legegyszerűbb megoldás, lévén, hogy nem használtunk semmiféle maszkolást és egyéb műveletet a logikai kapcsolatra. A BOB-nak tulajdonképpen önmaga a maszkja, így a kirakásnál a képbájt azon bitjei lesznek BOB-színűek, amelyeket a BOB mintája megad. A következő BOB, ha ugyanide kerül, természetesen a már előbb kirakott BOB-bal együtt fogja venni a képbájtot, tehát ez a BOB így takarni fogja a másikat.
5. Hogyan történik a BOB-ok kirakása a képre? A BOB kirakását az alábbi három utasítás végzi:

```
outsb  
mov    al,ah  
xchg   al,es:[di]
```

Az **outsb** kiviszi a maszkregiszterbe (\$3CE, \$08) a következő adatot. A DS:SI pointer a BOB Shape-re mutat, tehát a Shape következő bájtja kerül a maszkregiszterbe. A **mov al,ah** utasítás az **al**-be viszi a BOB színét. Az **xchg** végzi a tulajdonképpeni képmódosítást. Az utasítás beolvassa az **es:[di]** tartalmát, ezáltal a latch feltöltődik a képbájtjal. A csere hatására, amikor az **al** kiíródik a képre, a bitmaszk regiszter megadja a mintát, az **al** pedig a színt. Mivel kettes írás módot használunk, ezért a bitmaszk egyes helyein megjelenik az **al** szín, míg a nullás helyeken a latch tartalma, azaz az eredeti képbájt. Ez háromszor ismétlődik egy pixelsorban, majd 16-szor megismétlődik a teljes másolás, így az egész BOB láthatóvá válik.

Ennél a programnál is ugyanazt tanácsoljuk, mint az előbbinél: érdemes alaposan végigtanulmányozni, tisztába kerülni minden egyes részletével, mert ami ezután jön, az már feltételezi ennek a megoldásnak a megértését!

7.8. Többszínű, nem átfedő BOB-ok

Az eddig ismertetett BOB-megjelenítők egyszínű BOB-okkal dolgoztak, a most következő program viszont már 16 színű BOB-okat fog megjeleníteni és mozgatni. A „nem átfedő” jelző az első esethez hasonlóan azt takarja, hogy a mozgás csak abban az esetben lesz hibátlan, ha a BOB-ok nem ütköznek sem a háttérrel, sem egymással. Most teljesítjük ezt a feltételt, ezért nincs szükség a kettősbuffer-technikára. A program a következő újdonságokat fogja tartalmazni az előzőekhez képest:

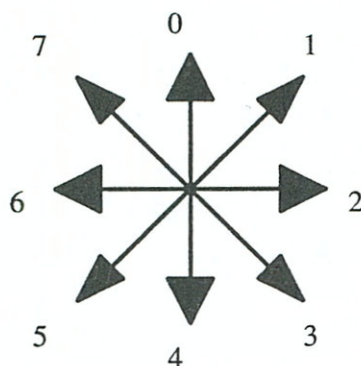
1. Táblázatos címszámításokat fogunk alkalmazni. A megjelenítés problémáinál volt szó arról, hogy az X,Y pixeles koordinátához tartozó képcímet kétféleképpen lehet kiszámítani: szorzással vagy táblázattal. Ebben a programban egy táblázat segítségével felgyorsítjuk a címszámítás műveletét. A képernyő minden rastersorának kezdőcímét tároljuk, így nem kell elvégezni a $80 \times Y$ szorzást, hanem csak az Y-adik kezdőcímet kell kivenni a táblázatból. Ugyancsak táblázatos módszerrel fogjuk meghatározni a Shape-ek címeit is.
2. Előre eltolt adatokkal fogunk dolgozni. Az összes mozgatható Shape-et elforgatjuk a program elején, és előállítjuk a nyolc lehetséges képet. Ezáltal a memóriaigény megnő ugyan, a megjelenítés viszont gyorsabb lesz.
3. Lesznek nem mozgó Shape-ek is. Ezeknek nincs különleges nevük, ezért ezeket önkényesen Brickeknek, tégláknak fogjuk nevezni. A Brick csak bájthatárra rakható ki, és nem végez mozgást. Tulajdonképpen a Brick felfogható úgy, mint egy $n(\text{mod } 8=0) \times m$ méretű színes grafikus karakter.
4. Az adatok tárolását színes bájtokon oldjuk meg. Ez azt jelenti, hogy mind a Brickek, mind a BOB-ok adatait a képmemóriában tároljuk, így két előnyös tulajdonságot tudunk kihasználni a megjelenítéskor:
 - a. egy képbájton négy CPU bájtot tudunk eltárolni,
 - b. egyszerre tudjuk mozgatni az egész színes bájtot.

A grafikus mintákat az AA00:0000 címre tesszük le. Tíz darab Brick lesz, ennek a memóriaigénye 10×32 , azaz 320 színes bájt. A Brick és a BOB is 16×16 pixeles lesz. Ha nem a képmemóriában tárolnánk az adatokat, akkor egy Brick $4 \times 32 = 128$ bájtot igényelne, míg egy Shape eltárolásához — mivel előre eltolt állapotban van — $8 \times 4 \times 48 = 1536$ bájtra lenne szükség. Így azonban egy Brick 32, míg egy shape 384 színes bájtot igényel. Két Shape között tehát 384 bájt a távolság, míg egy Shape pixeles eltolásai között 48 bájt. A cím ezért a

$$\text{Shape kezdőcím} = 384 \times \text{Shape-szám} + 48 \times (X \text{ AND } 7) + \text{abszolút kezdőcím}$$

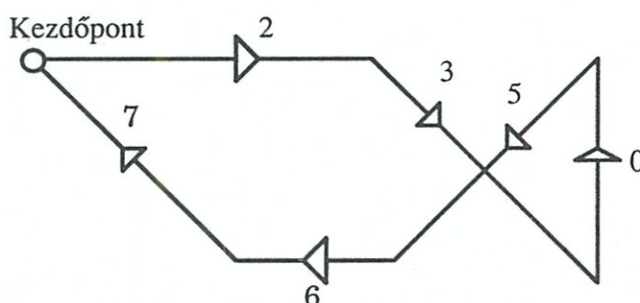
képlettel számítható. Mindkét szorzást táblázattal oldjuk meg. Az *abszolút kezdőcím* az AA00 szegmensben belüli eltolást adja meg. A Brick adatok 320 bájtot foglalnak le, ezért ezek az AA00:0000 címen, míg a Shape adatok az AA00:0140 címen érhetők el.

5. Előre programozott mozgáspályákat alkalmazunk. Ez a fajta mozgás kötött pályát szab meg a BOB-nak, amelyről az nem térhet le. A programozott mozgás lényege a következő: definiált irányokkal dolgozunk. Az irányok számát tekintve nincs korlátozás, de ebben az esetben csupán a nyolc kitüntetett irányt alkalmazzuk:



39. ábra Elmozdulások a programozott mozgás során

Ezek után egy adattömbbel meghatározzuk a mozgáspályát. Ez egy egyszerű irány—lépés összerendelés lesz: adott irányhoz adott lépésszám. Nézzünk egy egyszerű példát:



40. ábra Egy mozgásprogram

A nyilakkal jelzett irányban mozgunk végig a pályán. Látható, hogy az irányokat és a lépéseket úgy kell megválasztani, hogy a mozgás a kezdőpontba térjen vissza, mégpedig pontosan ugyanoda, különben a BOB valamelyik irányban szépen kikacsázik a képernyőről. Ennek a huroknak a leírásához (ha az irányt és a lépést is bájtokon tároljuk) 6×2 bájt memória kell, plusz egy jelzőbájt az adathalmaz végére, amely jelzi, hogy ismét előlről kell kezdeni a mozgásprogramot.

Nagyjából tehát ennyi módosítás lesz az előző megjelenítőkhöz képest. Maga a megjelenítő egyszerűsödni fog, de az előkészületek több feladatot jelentenek. Lássuk a megvalósítást!



BOB2.PAS

```
{ $G+}
Uses Crt;
```

A *BOBvar* rekord mezőiben a mozgásprogramhoz szükséges két változó is helyet kap. A *Prg* adja a BOB-hoz tartozó programszámot, míg a *Ppt* egy word hosszúságú mutató a mozgásprogram következő irány—lépés adatpárosára.

```

Type BOBvar = record
    X,Y : word;      { [bp+0],[bp+2] }
    S   : word;      { [bp+4]       }
    Ir  : byte;      { Irány (0..7) }
    Lp  : byte;      { Lépésszám   }
    Prg : byte;      { Mozgásprogram }
    Ppt : word;      { Programmutató }
end;

```

```

MemPtr = ^Memory;
Memory = Array[ 0..65534] of byte;

```

Az általános állandók és változók hasonlóak az előző programokéhoz.

Const

```

{ Egy BOB rekord hossza }
sizeofBOB = sizeof(BOBvar);

SegA000 : word = $a000;
SegAA00 : word = $aa00;

{ 11 db BOB lesz }
BOBnum = 11;
{ 640x480-as képet használunk, ezért a logikai hossz 80 }
Lsize = 80;

{ Bitplane számok a $3c4,$02 regiszter számára }
Planes : Array [ 0..3] of word=($0102,$0202,$0402,$0802);

{ Mozgásprogramok }
MovePrg : Array [ 0..80] of byte=(
    $ff, 2,200, 6,200,
    $ff, 4,200, 0,200,
    $ff, 2,50, 3,50, 5,50, 6,50, 7,50, 1,50,
    $ff, 2,130, 4,130, 6,130, 0,130,
    $ff, 3,50, 5,50, 7,50, 1,50,
    $ff, 3,50, 0,50, 5,50, 0,50,
    $ff, 3,50, 5,50, 3,50, 7,50, 1,50, 7,50,
    $ff, 2,50, 3,50, 4,10, 5,50, 6,50, 7,50, 0,10, 1,50,
    $ff );

```

A *MovePrg* tömb az előre definiált pályákat tartalmazza. Nyolc pályát tárolunk itt, és mindegyik egy kezdőpontba visszajutó mozgást ír le.

```

{ Az irányokhoz tartozó elmozdulások }
dX : Array [ 0..7] of shortint=(0,1,1,1,0,-1,-1,-1);
dY : Array [ 0..7] of shortint=(-1,-1,0,1,1,1,0,-1);

```


A BOB-ok koordinátáit most abszolút pozíciókban kell megadni:

```
{ A BOB-ok kezdőadatai: koordináták és mozgásprogram számok }
Xcr      : Array [ 1 .. BOBnum] of word=(
           40,200,580,480,330,30,50,320,100,600,550);
Ycr      : Array [ 1 .. BOBnum] of word=(
           40,40,40,100,150,200,440,320,280,200,330);
Prgs     : Array [ 1 .. BOBnum] of byte=(
           3,0,1,5,2,6,0,7,0,1,4);
```

Var

```
i,j,k,l : integer;
w        : word;
f        : file;
fsize    : longint;
BOBS     : Array [ 1..BOBnum] of BOBvar; { BOB adatterület }
P        : MemPtr; { A mintákra mutató
                    pointer }
Rbuff    : Array [ 0..191] of byte; { Forgatóbuffer }

Sadrtab  : Array [ 0..7] of word; { Mintacím eltolások }
Xpixtab  : Array [ 0..7] of word; { Pixeles eltolások }
Addrtab  : Array [ 0..479] of word; { Képcímtáblázat }
```

Ez a három táblázat a címszámításokhoz szükséges. Az *Sadrtab* a Shape-ek kezdőcímeit tárolja, az *Xpixtab* a pixeles eltolás miatti korrekciót, míg az *Addrtab* a raszter-sorok kezdőcímeit a képcímszámításhoz.

```
{ A BOB-ok betöltése a heapbe }
```

Procedure LoadBOBS;

Begin

```
assign(f,'bob1.dat');
reset(f,1);
fsize:=filesize(f);
getmem(P,fsize);
blockread(f,P^,fsize);
close(f);
```

End;

A *LoadBOBS* behívja a lemezről a *Brick* és a *Shape* adatokat és a heapben tárolja azokat. Innen kell majd bemásolni azokat a képmemóriába az AA00:0000 címre.

{ Egy BOB bemásolása a forgatóbufferbe }

Procedure MemToRbuff(m:MemPtr); **Assembler;**

Asm

```

    cld
    mov  bx,DS
    mov  ES,bx
    lea  di,Rbuff          { ES:DI az Rbuff-ra mutat          }
    lds  si,m              { DS:SI pedig az adatokra          }
    mov  cx,4*16           { 4*16 sort kell átmásolni          }
    xor  al,al             { AL = 00                            }

```

@move:

```

    movsw          { BOB adat                                }
    stosb          { 0 a forgatás miatt                      }
    loop @move
    mov  DS,bx      { Az eredeti DS visszatöltése            }

```

End;

A *MemToBuff* eljárás egy Shape adatot másol a heapből a forgatóbufferbe.

Procedure RotRbuff; **Assembler;**

Asm

```

    mov  cx,192          { Az Rbuff mérete 4*48 bájt          }
    clc
    lea  bx,Rbuff        { A BX-be tesszük a kezdőcímet        }

```

@rot:

```

    rcr  byte [bx],1     { Egy bájt elforgatása                }
    inc  bx              { A buffer következő bájtja          }
    loop @rot

```

End;

A *RotBuff* egy pixellel jobbra tolja a bufferben lévő Shape-et.

A következő, *Convert_Datas* rutin végzi a Shape-ek színes adattá való konvertálását, valamint a képmemória feltöltését az AA00:0000 címtől kezdve.

Procedure Convert_Datas;

Begin

```

    LoadBOBS;          { A BOB-ok betöltése a heapbe }
    { Az első 10 adat fix mintákat tartalmaz (Bricks), ezek nem fognak
    mozogni }

```

for i:=0 **to** 9 **do**

Begin

for j:=0 **to** 3 **do** { Színes bájtta alakítja a Shape-t }

Begin

portw[\$3c4]:=Planes[j]; { Az írni kívánt plane kiválasztása }

move(P^[i*128+j*32], MEM[SegAA00:i*32], 32);

End;

End;


```

{ A következő 8 Shape mozgó BOB lesz, ezért ezeket forgatni is
  kell }
for i:=10 to 17 do
  Begin
    MemToRbuff(Addr(P^[ i*128 ])); { Az adat a bufferbe kerül }
    for l:=0 to 7 do { 8 eltolt mintát kell készíteni }
      Begin
        if l>0 then RotRbuff; { Forgatás az 1..7 esetekben }
        for j:=0 to 3 do { Kiviszi az AA00:0000 címre }
          Begin
            portw[ $3c4 ] := Planes[ j ] ;
            portw[ $3ce ] := j*256+4;
            move(Rbuff[ j*48 ], MEM[ SegAA00:l*48+(i-10)*384+320 ], 48);
          End; { for j }
        End; { for l }
      End; { for i }
    End; { for i }

    freemem(P, fsize); { Felszabadítja a heap lefoglalt részét }
  End;

```

A *Convert_Datas* után a heap felszabadul, és az AA00:0000 címen ott van a tíz darab Brick adat valamint az AA00:0140 címen a nyolc darab eltolt Shape adat.

```

Procedure _640x480_c16; Assembler;
Asm
  mov    ax, 12h      { 640x480/16-os mód }
  int     10h
  mov     dx, 03ceh
  mov     ax, 0106h
  out     dx, ax      { 128K video-RAM engedélyezése }
End;

```

A *_640x480_c16* bekapcsolja a grafikus módot, és 128 Kb-ja árra állítja a videomemória méretét.

```

Procedure Brick(X,Y,Z:word); Assembler;
Asm
  mov     dx, DS
  lea     bx, AddrTab { A BX a képcímtáblázatra mutat }
  mov     di, Y       { Az Y koordináta a DI-be kerül }
  shl     di, 1       { Az Y-t megszorozzuk kettővel (mert word
                      adatok) }
  mov     di, [bx+di] { Kivesszük a táblázatból a megfelelő
                      képcímet }
  shl     X, 1       { Az X szorzása kettővel: 0..39 koordináta
                      lehet }
  add     di, X       { Címszámítás a táblázatos módszerrel,
                      DI = képcím }

```

```

mov     si,Z           { Az SI-be a minta száma }
shl     si,5           { A minta címének kiszámítása 32*Z }
mov     ES,SegA000     { ES:DI mutat a képre }
mov     DS,SegAA00     { DS:SI pedig a Shape színes bájtjaira }
cld
mov     cx,16          { A Shape 16 pixelsor magas }
@put:
movsb
movsb           { Két színes bájt átmásolása }
add     di,Lsize-2
loop    @put       { A teljes Shape kirakása }
mov     DS,dx
End;

```

A *Brick* eljárás az *X,Y* pozícióba kiteszi a *Z*-edik *Brick* adatot. Az *X* ebben az esetben 16 pixelenként lépked, azaz az értéke 0..39 lehet. Az *Y* pixeles.

A mozgó BOB-ok megjelenítése a *ShowBOBS* rutinnal:

Procedure ShowBOBS; **Assembler;**

Asm

```

push    bp
lea     bp,BOBS        { A DS:[BP] fog a BOB adatokra mutatni }
mov     dx,DS          { A DX tárolja az eredeti adat-
                        szegmenset }
mov     es,SegA000     { Az ES a képre mutat }
mov     cx,BOBnum      { Ennyi BOB van }
@main:
push    cx             { Tároljuk a BOB-számlálót }
mov     DS,dx          { Az adatok az eredeti DS-ben vannak }
{ Az első lépésben meghatározzuk a forrásadat kezdőcímét }
lea     bx,SadrTab     { A BX a Shape kezdőcím táblázatra
                        mutat }
mov     si,DS:[bp+4]   { A Shape kezdőcímét határozza meg az
                        S mező alapján }
shl     si,1
mov     si,[bx+si]     { Az SI most az eltolást tartalmazza }
lea     bx,Xpixtab     { A pixeles eltolást is bele kell
                        számítani }
mov     di,DS:[bp]     { A DI-be tesszük az X koordinátát }
mov     ax,di          { Az AX tárolja az X-et későbbre }
and     di,7           { Az X alsó három bitje adja a pixeles
                        eltolást }
shl     di,1           { A táblázat indexelése miatt szorozzuk
                        kettővel }
add     si,[bx+di]     { Az eltoláshoz hozzáadja a korrekciót }

```



```

{ Most kiszámítjuk a képcímet a táblázatos módszerrel }

lea    bx,AddrTab  { A BX a képcímtáblázatra mutat }
mov    di,DS:[bp+2] { A DI-be az Y koordináta }
shl    di,1        { Y*2, mert a táblázat word adatokat
                    tartalmaz }
mov    di,[bx+di]   { Az Y-hoz tartozó cím a DI-be kerül }
shr    ax,3         { Eltávolítjuk az X alsó három bitjét,
                    osztás 8-cal }
add    di,ax        { ES:DI tehát most a képernyőre mutat }

mov    DS,SegAA00   { DS:SI pedig a forrás színes bájtokra }
cld

{ Most ki kell tenni a BOB-ot }

mov    cx,16        { Egy BOB magassága 16 pixelsor }
@put:
movsb
movsb
movsb               { Három színes bájt átmásolása }
add    di,Lsize-3
loop   @put         { Az egész BOB megjelenítése }

pop    cx
add    bp,sizeofBOB { A BP a következő BOB adataira mutat }
loop   @main        { Az összes BOB-ot megjeleníti }

mov    DS,dx
pop    bp
End;

```

Egy adott számú mozgásprogram megkeresése a *Prog* függvénnel:

```

Function Prog(b:byte):word;Assembler;
Asm
  cld
  mov    dx,DS
  mov    ES,dx      { Az ES ráállítása az adatszegmensre }
  lea    di,moveprg
  mov    ax,0ffffh   { 0fffh-t kell keresni, ez az AL-ben van }
@c:
  inc    ah          { A program számát az AH adja }
  mov    ch,al       { A CX = 0fff??h mindenképpen elég a
                    kereséshez }
  repne scasb        { Megkeresi a következő programot }
  cmp    ah,b        { Megtalálta már a keresett programot ?? }

```

```

jnz    @c                { Folytatja, ha még nem találta meg      }
mov     ax,di             { A program eltolása az AX-be           }
sub     ax,offset MovePrg
      { Az AX megadja a program indexét a MovePrg tömbhöz        }
End;

```

A BOB-ok mozgatása előre meghatározott mozgásprogram szerint történik:

```

Procedure MoveBOBS;
Begin
  for i:=1 to BOBnum do { Az összes BOB-ot mozgatja           }
    with BOBS[ i] do
      Begin
        dec(Lp);           { A lépésszám csökkentése           }
        if Lp=$ff then
          Begin
            inc(Ppt);       { A következő irány                }
            Ir:=MovePrg[ ppt] ;
            if Ir=$ff then { Ha ff, akkor előlről kell kezdeni a }
                          { programot                             }
              Begin
                Ppt:=Prog(Prg); { Megkeresi a program elejét    }
                Ir:=MovePrg[ Ppt] ; { Legelső irány          }
                Lp:=MovePrg[ Ppt+1] ; { Legelső lépés          }
                inc(Ppt);
              End else
              Begin
                inc(Ppt);
                Lp:=MovePrg[ Ppt] ; { Következő lépés      }
              End;
            End;
          { Elmozdítja a BOB-ot az irány szerint }
          inc(X,dX[ Ir] );
          inc(Y,dY[ Ir] );
        End;
      End;

```

Az alapkép kirajzolása a Brickek kirakásával történik:

```

Procedure Gen_Picture;
Begin
  { A felső és az alsó keret meghúzása }
  for i:=1 to 38 do
    Begin
      Brick(i,0,3);
      Brick(i,29*16,3);
    End;

```



```

{ A bal és a jobb szélső keret meghúzása }
for i:=1 to 28 do
  Begin
    Brick(0,i*16,5);
    Brick(39,i*16,5);
  End;
{ A keret négy sarkának kirakása }
Brick(0,0,2);
Brick(39,0,4);
Brick(0,29*16,6);
Brick(39,29*16,7);
{ Pályaelemek }
for i:=0 to 5 do
  for j:=0 to 5 do
    Brick(i+4,(j+4)*16,8);
for i:=0 to 30 do Brick(5+i,300,9);
for i:=0 to 15 do Brick(35,(3+i)*16,9);
for i:=0 to 4 do
  Begin
    Brick(10+i,400,1);
    Brick(20+i,200,1);
    Brick(30,i*16+350,1);
  End;
End;

```

BEGIN

A főprogramban el kell végezni a kezdeti beállításokat:

```

{ A grafikus mód bekapcsolása }
_640x480_c16;

{ A címszámító táblázatok feltöltése }
{ Két mozgó Shape között 8*48=384 az eltolás, az adatok kezdőcíme
320 }
for w:=0 to 7 do SadrTab[ w ]:=w*384+320;
{ Két eltolt adat között 48 bájt a címkülönbség }
for w:=0 to 7 do XpixTab[ w ]:=w*48;
{ A képernyő 480 sorának kezdőcímeit számítjuk ki }
for w:=0 to 479 do AddrTab[ w ]:=w*Lsize;

{ Betölti és konvertálja az adatokat, kialakítja a színes bájtokat }
Convert_Datas;
{ A bitmaszk nulla, ezért másolni fog }
portw[ $3ce ]:=$0008;
{ A bitplane-eket írásra engedélyezni kell }
portw[ $3c4 ]:=$0f02;

```

```

{ A BOB adatokat fel kell tölteni a megadott értékekkel }
for i:=1 to BOBnum do
  with BOBS[ i] do
    Begin
      S:=i and 7;          { A Shape száma az i-től függ }
      X:=Xcr[ i] ;        { A koordináták adottak }
      Y:=Ycr[ i] ;
      Prg:=Prgs[ i] ;      { A mozgásprogram adott }
      Ppt:=Prog(Prg);      { Az első adatok megkeresése }
      Ir:=MovePrg[ Ppt] ;  { Első irány }
      Lp:=MovePrg[ Ppt+1] ; { Első lépésszám }
      inc(Ppt);
    End;
  { Előállítja a háttérképet }
  Gen_Picture;

  { Mozgatási és megjelenítési főciklus }
  repeat
    MoveBOBS;      { Elmozdítja a BOB-okat }
    ShowBOBS;      { Megjeleníti a BOB-okat }
    delay(10);     { Késleltetés }
  until keypressed;
  while keypressed do readkey;

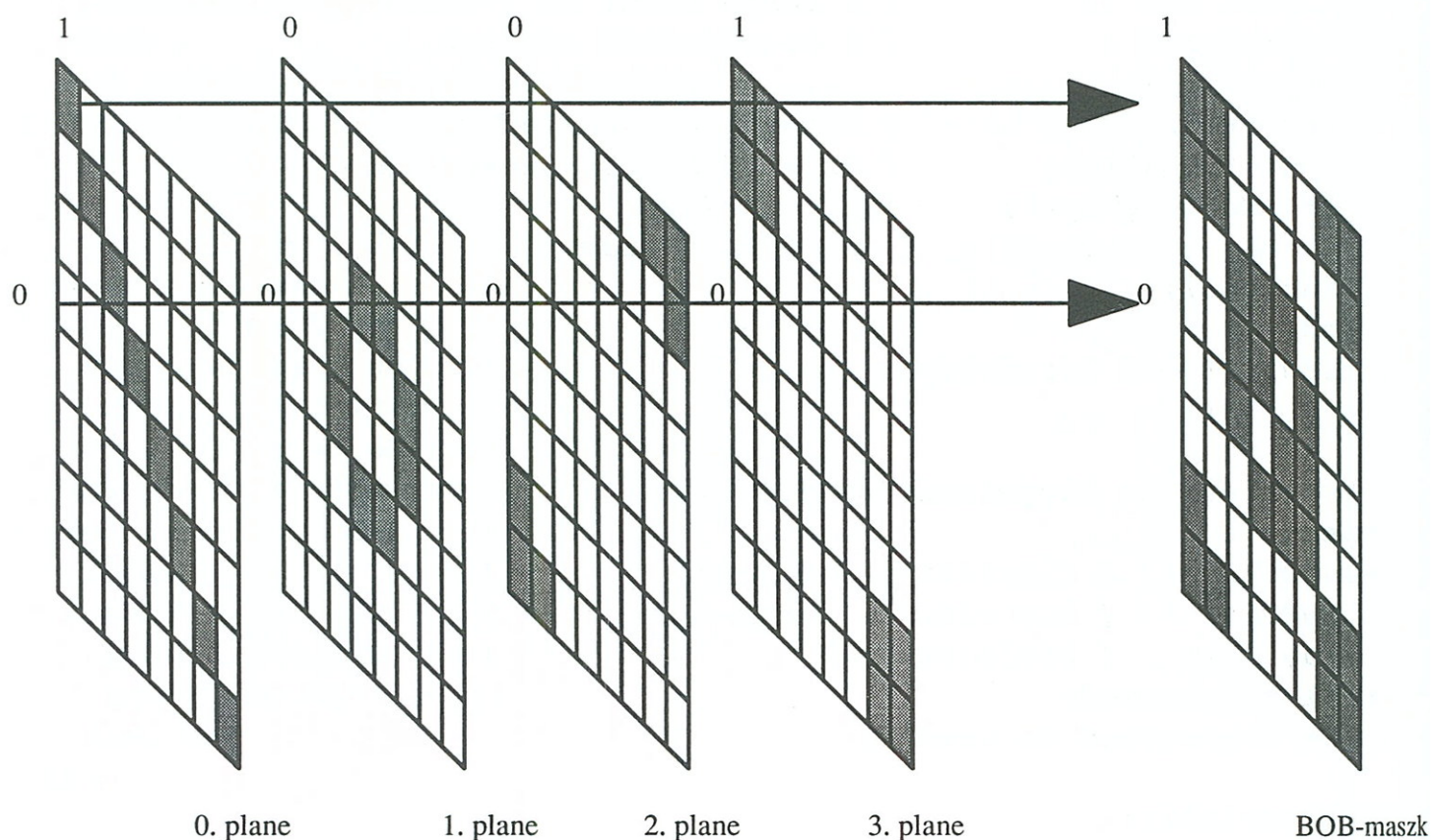
  textmode(c80);
END.

```

7.9. Többszínű, átfedő BOB-ok

A többszínű, átfedő BOBok — mint ahogyan azt bizonyára az Olvasó is sejti — igénylik a legbonyolultabb megjelenítési algoritmust. A BOB-ok tizenhat színűek, és egymással, valamint a háttérrel ütközhetnek. Nyilvánvaló, hogy a kettősbuffer-módszert kell alkalmaznunk, csak hogy a BOB-ok színesek, ezért a BOB és a kép logikai kapcsolata bonyolultabb, mint az egyszínű BOB-oknál. A BOB — a képhez hasonlóan — négy bitplane-ből áll. A BOB-ot tehát jó lenne a képmemóriában, színes bájtokon tárolni, de sajnos a VGA kártya nem tud két színes bájtot úgy összemontírozni, ahogyan arra nekünk szükségünk volna, ezért a BOB-okat mégis a normál memóriában tároljuk el. A feladat világos: a BOB megfelelő bitplane-jeit rá kell másolni a kép megfelelő bitplane-jeire. Ez azonban még nem egyértelmű összerendelés, mert nem tudjuk, hogy egy adott plane-en mely biteket takar a BOB, és mely biteket hagy láthatóan. Ennek az egyértelmű meghatározására szolgál a BOB-maszk. A BOB-maszk nem más, mint egy „árnykép”, amely mint éles kontúrt ábrázolja az egész BOB-ot, és egyértelműen megmutatja, hogy hol van szükség a képadatra és hol

a BOB-adatra. A maszk előállítására több lehetőség is kínálkozik. A maszkot általában ugyanúgy megtervezik, ahogyan a többszínű Shape-et, de mi most közvetlenül a Shape-ből állítjuk majd elő.



41. ábra A BOB-maszk előállítása

Látható, hogy a maszk nem más, mint a bitplane-ek egyszerű VAGY (OR) kapcsolata. A maszk segítségével már meg tudjuk határozni, hogy hol jelenjen meg a BOB adat (ahol a maszk 1-es bitet tartalmaz), és hol jelenjen meg a képadat (ahol a maszk 0-s bitet tartalmaz). Tehát a képre háttérszínnel (feketével) ki kell vinni a maszkot, majd a képet egyszerűen VAGY (OR) kapcsolatba kell hozni a BOB-bal. A VAGY kapcsolat azért felel meg nekünk, mert a fekete maszk kirajzolása tulajdonképpen egy ÉS (AND) kapcsolatot jelent a maszk és a kép között, tehát azokon a helyeken marad csak meg a képadat, ahol biztosan nem lesz BOB adat. Ezért egyszerűsödik le a képre rajzolás egy szokásos VAGY kapcsolatra. A VAGY kapcsolat meghagyja a kép azon bitjeit, amelyeket a maszk nem tüntetett el, és kirakja a BOB azon bitjeit, amelyek biztosan a maszk által „megtisztított” helyre kerülnek. A megjelenítés tehát kiegészül a maszk kirajzolásával, és a BOB-ok kirakása módosul, mivel most minden plane-t külön kell kivinni a képre. A címszámításokat az előző esethez hasonlóan táblázattal oldjuk meg, és a Shape adatokat előre eltologatjuk. Így egy Shape $4 \cdot 48 \cdot 8 = 1536$ bájt memóriát igényel. A program egyébként ötvözi az előző három program különböző megoldásait. A mozgatáshoz két mozgató algoritmust fogunk használni: egyet a moz-

gásprogram alapján és egy másikat, a véletlenszerűen mozgató algoritmust. A programban sok rutint az előző programokból veszünk át, ezért ezeknek a magyarázatát most már nem részletezzük.



BOB3.PAS

{ \$G+}

Uses Crt;

Type

{ Egy BOB adathalmaza }

BOBvar = **record**

X,Y :	word;	{ Koordináták [bx+0], [bx+2]	}
S :	word;	{ Shape-szám [bx+4]	}
AdA :	word;	{ Képcím az A képen [bx+6]	}
AdB :	word;	{ Képcím a B képen [bx+8]	}
Sad :	word;	{ [bx+10] Shape cím	}
Mad :	word;	{ [bx+12] maszk cím	}
Ir :	byte;	{ Irány	}
Lp :	byte;	{ Lépés	}
Prg :	byte;	{ Mozgásprogram	}
Ppt :	word;	{ Programmutató	}
end;			

A BOB rekord kiegészült két mezővel:

Sad: a Shape kezdőcíme, amelyet a rutin az S mintaszámból számol ki.

Mad: a maszk kezdőcíme, amelyet szintén az S alapján határozzunk meg.

{ Memória- és memóriamutató típus }

MemPtr = ^Memory;

Memory = **Array**[0..65534] **of** byte;

Const

BOBsize = SizeOF(BOBvar);

SegA000 : word = \$a000;

Lsize = 40; { A képernyő logikai hossza }

BOBnum = 16; { Tizenhat BOB fog mozogni }

{ Bithelyek a Putpixel rutinhoz }

Bits : **Array** [0..7] **of**

byte=(\$80,\$40,\$20,\$10,\$08,\$04,\$02,\$01);

{ Elmozdulások a 8 irányhoz }

dX : **Array** [0..7] **of** shortint=(0,1,1,1,0,-1,-1,-1);

dY : **Array** [0..7] **of** shortint=(-1,-1,0,1,1,1,0,-1);


```
{ A 320x240-es felbontáshoz tartozó CRTC (3d4,3d5) regiszter
    értékek }
```

```
_Data_240 : Array [ 0..$17] of byte=
    ($2d,$27,$28,$90,$2b,$80,$0b,$3e,
    $00,$c0,$00,$00,$00,$00,$00,$00,
    $ea,$8c,$df,$14,$00,$e7,$04,$e3);
```

```
Inactive : word = $A258;
Scrbuff  : word = $A4B0;
Act       : word = $0002;
InacScr   : word = $2580;
```

```
{ Mozgásprogram }
```

```
MovePrg : Array [ 0..9] of byte=($ff, 2,240, 4,150, 6,40,
7,50,$ff);
```

Var

```
i,j,k,l : integer;
w        : word;
b        : byte;
f        : file;
fsize    : longint;
BOBS     : Array [ 1..BOBnum] of BOBvar; { BOB adatterület }
P        : MemPtr; { A mintákra mutató
                    pointer }

Rbuff    : Array [ 0..191] of byte; { Forgatóbuffer }

Sadrtab  : Array [ 0..15] of word; { Mintacím eltolások }
Madrtab  : Array [ 0..15] of word; { Maszkcím eltolások }
Xpixtab  : Array [ 0..7] of word; { Pixeles eltolások
                                   (maszk) }

Xpixtab2: Array [ 0..7] of word; { Pixeles eltolások
                                   (Shape) }

Addrtab  : Array [ 0..239] of word; { Képcímtáblázat }

Shapes   : Array [ 0..24575] of byte; { BOB-minták }
Masks    : Array [ 0..6143] of byte; { Maszkminták }
```

```
{ A BOB-ok betöltése a heapbe }
```

Procedure LoadBOBS;

Begin

```
assign(f,'bob2.dat'); { A bob2.dat fájl a lemezmellékleten
}
```

```
reset(f,1);
```

```
fsize:=filesize(f);
```

```
getmem(P,fsize);
```

```
blockread(f,P^,fsize); { A fájlt betölti a heapbe }
```

```
close(f);
```

End;

{ Egy BOB bemásolása a forgatóbufferbe }

Procedure MemToRbuff(m:MemPtr); **Assembler;**

Asm

```

  cld
  mov  bx,DS
  mov  ES,bx
  lea  di,Rbuff          { ES:DI az Rbuff-ra mutat      }
  lds  si,m              { DS:SI pedig az adatokra      }
  mov  cx,4*16           { 4*16 sort kell átmásolni     }
  xor  al,al

```

@move:

```

  movsw          { BOB adat                          }
  stosb          { 0 a forgatás miatt                  }
  loop @move
  mov  DS,bx

```

End;

{ A bufferadat eltolása egy pixellel jobbra }

Procedure RotRbuff; **Assembler;**

Asm

```

  mov  cx,192           { A buffer hossza 4*48 bájt      }
  clc
  lea  bx,Rbuff         { A buffer kezdőcíme a BX-be      }

```

@rot:

```

  rcr  byte [bx],1      { Minden bájtot eltol egy bittel jobbra }
  inc  bx
  loop @rot

```

End;

{ Az adatok konvertálása és maszkszámolás }

Procedure Convert_Datas;

Begin

LoadBOBS; { Adatbetöltés }

k:=0;

for i:=0 **to** 15 **do**

Begin

MemToRbuff(Addr(P^[i*128])); { Az adat a bufferbe kerül }

for l:=0 **to** 7 **do** { 8 eltolt mintát kell készíteni }

Begin

if l>0 then RotRbuff; { Eltolás a bufferben }

move(Rbuff[j*48] ,Shapes[k] ,48*4); { A Shapes tömb
feltöltése }

inc(k,48*4); { A Shapes tömb indexének növelése
(192) }

End; { for l }

End; { for i }

freemem(P,fsize); { Felszabadítja a heap lefoglalt részét }


```

{ A maszkok előállítása egyszerű OR kapcsolattal }
for i:=0 to 127 do
  Begin
    for j:=0 to 47 do
      Begin
        b:=Shapes[ i*192+j+0*48] or      { Minden plane-t össze-OR-ol, }
          Shapes[ i*192+j+1*48] or
          Shapes[ i*192+j+2*48] or
          Shapes[ i*192+j+3*48] ;
        Masks[ i*48+j] :=b;              { és ez lesz a maszk          }
      End;
    End;
  End;

{ A 320x240/c16-os mód beállítása }
Procedure _320x240_c16; Assembler;
Asm
  .... az előző részekben már ismertetett eljárás ... (BOB1.PAS)
End;

{ Képinicializálás, a B és a C kép feltöltése }
Procedure Screen_Init; Assembler;
Asm
{
  Memóriaafelosztás:
  "A" kép : A000:0000
  "B" kép : A000:2580   normalizálva : A258:0000
  "C" kép : A000:4B00   normalizálva : A4B0:0000
}
  .... az előző részekben már ismertetett eljárás ... (BOB1.PAS)
End;

{ Egy pont kigyújtása }
Procedure PutPixel(X,Y:word;color:byte); Assembler;
Asm
  .... az előző részekben már ismertetett eljárás ... (BOB1.PAS)
End;

```

A programban az eddigi eljárások már ismertek voltak korábbról. A következő *ShowBOBS* azonban lényeges különbségeket fog mutatni.

```

(*****)
(*  A BOB-ok megjelenítése  *)
(*****)
Procedure ShowBOBS; Assembler;
Asm
  push  DS
  push  bp

```

```

mov     dx,03ceh
mov     ax,0205h      { Mode 2 írás módot használ a kép
                      helyreállításához }

out     dx,ax
mov     ax,0003h      { A logikai kapcsolat normál move }
out     dx,ax
mov     ax,0008h      { A bitmaszk nulla, így másolni fog }
out     dx,ax

mov     bp,DS         { A BP az eredeti adatszegmenset tárolja }
mov     ES,Inactive   { Az ES az inaktív képre mutat }
mov     dx,ScrBuff    { A DX a bufferkép szegmensét tárolja }

lea     bx,BOBS        { A BX a BOB adatokra mutat }
add     bx,Act         { Ez a korrigáló érték 0/2 (AdA vagy AdB) }
cld

{ Az első lépésben helyreállítja az inaktív képet a címek alapján }

mov     ax,BOBnum      { Ennyi BOB van }
@restore:
mov     DS,bp          { A BOB adatok az eredeti DS-ben vannak }
mov     di,[bx+6]      { Kiveszi a megfelelő címet }
add     bx,BOBsize     { A BX a következő BOB adatra mutat }
mov     si,di          { Az offsetek megegyeznek }
mov     DS,dx          { A DS-t a "C" bufferképre állítja }
mov     cx,16

@res:
movsb                     { !! színes bájtok másolása !! }
movsb
movsb
add     di,Lsize-3
add     si,Lsize-3
loop   @res              { Egy BOB alatti terület helyreállítása }
dec     ax
jnz     @restore         { Minden BOB-ot letöröl }

{ Az új pozícióhoz kiszámítja az új adatokat }

```

Mivel a fő megjelenítési ciklus túlságosan összetett, ezért két részre választjuk. Az első rész kiszámítja a koordinátákból a képcímet, valamint a Shape-számból a Shape és a maszk kezdőcímet. Ezek az adatok rendre az *AdA(AdB)*, *Sad* és *Mad* mezőkbe kerülnek, ahonnan a második megjelenítő rész már ezekkel a kész címekkel fog dolgozni.


```

mov     DS,bp                { A DS-t visszaállítjuk az eredeti
                               értékére }
lea     bx,BOBS              { A BX a BOB-ok adataira mutat }
mov     cx,BOBnum            { Ennyi BOB van }
@main1:
mov     si,Act
lea     bp,AddrTab           { Képcímszámítás táblázat alapján }
mov     di,[ bx+2]    { Y }
shl     di,1                 { *2, mert word adatokat használunk }
mov     di,DS:[ bp+di]       { Az Y-hoz tartozó rasztersor cím }
mov     ax,[ bx]    { X }
mov     dx,ax                { DX = X, tárolás későbbi használatra }
shr     ax,3                 { Az X alsó három bitje most nem kell }
add     ax,di
mov     [ bx+si+6] ,ax        { Az AdA/AdB-be az új képcím }
lea     bp,SadrTab           { A Shape kezdőcím kiszámítása }
mov     si,[ bx+4]    { S }
shl     si,1                 { *2, mert word adatokat használunk }
mov     si,DS:[ bp+si]
lea     bp,XpixTab2          { A címet a pixeles eltolás is
                               befolyásolja }

mov     di,dx
and     di,7                 { Most kell az X alsó három bitje }
shl     di,1                 { *2, mert word adatokat használunk }
mov     di,DS:[ bp+di]
add     si,di
mov     [ bx+10] ,si         { Shape cím a Sad-ba }
lea     bp,XpixTab           { A maszk kezdőcímének kiszámítása }
mov     di,dx
and     di,7                 { Az X alsó három bitje }
shl     di,1
mov     di,DS:[ bp+di]       { A maszk pixeles eltolása }
lea     bp,MadrTab           { A maszk abszolút címei }
mov     si,[ bx+4]    { S }
shl     si,1
mov     si,DS:[ bp+si]
add     si,di
mov     [ bx+12] ,si         { Maszkcím a Mad-ba }
add     bx,BOBsize
loop    @main1               { Minden BOB adatát kiszámolja }

{ A második lépésben a BOB-ok kirakása a képre }

lea     bx,BOBS              { A BX a BOB adatokra mutat }
mov     cx,BOBnum            { Ennyi BOB van }
@main2:
push    cx

{ Először a maszkot kell kirakni feketével minden plane-re }

```

A maszk kirakása **Mode 2** írás módban történik. A következő regisztereket kell módosítani:

\$3CE, 05: olvasás/írás mód regiszter
 \$3CE, 03: *Data Rotate & Fn Select* regiszter, mivel a logikai kapcsolat ekkor normál *move*, míg a BOB rajzolásakor *or*
 \$3CE, 08: bitmaszk regiszterrel adjuk meg a maszk bitmintáját

```

mov    dx,03ceh
mov    ax,0205h      { Kettes írásmód a maszkok kirajzolásához }
out    dx,ax
mov    ax,0003h      { move művelet a maszkok kirajzolásához }
out    dx,ax
mov    al,08h
out    dx,al        { A bitmaszk regiszter megcímzése }
inc    dx            { DX = $3CF, bitmaszk adatregiszter }

mov    si,Act        { AdA/AdB kiválasztás (0/2) }
mov    di,[ bx+si+6] { Képcím }
mov    si,[ bx+12]   { Maszkcím }

push   di            { A címre még szükség lesz }
xor    al,al         { maszkszín: fekete (háttér) }
mov    cx,16         { A maszk 16 pixel magas }

```

A *@putmask* ciklus kirajzolja a maszkot a képre.

```

@putmask:
  outsb      { A bitmaszk regiszter feltöltése }
  mov    ah,ES:[ di]  { A latch feltöltése }
  stosb     { Képbájt módosítás }
  outsb
  mov    ah,ES:[ di]  { 2. képbájt }
  stosb
  outsb
  mov    ah,ES:[ di]  { 3. képbájt }
  stosb
  add    di,Lsize-3
  loop   @putmask     { Egy BOB-maszk kirakása a képre }
  dec    dx           { DX most ismét 03CEh }

  pop     di          { A képcím eredeti értéke vissza a DI-be }
  mov     si,[ bx+10] { Shape-kezdőcím az SI-be }

```


Most következnek a BOB-ok kirakása bitplane-enként. Ezt **Mode 0** írás módban kell elvégezni, ezért néhány regisztert ismét át kell írni:

```

mov     dx,03ceh
mov     ax,0005h      { Mode 0 írás mód a BOB bitplane-ek
                       kimásolásához }

out     dx,ax
mov     ax,1003h
out     dx,ax      { OR logikai kapcsolat a CPU és a latch
                       között }

mov     ax,0ff08h
out     dx,ax      { A bitmaszknak 255-nek kell lennie }
mov     dx,03c4h    { Plane írásmaszok regiszter }
mov     ah,01h

{ Ez a ciklus jeleníti meg a BOB négy bitplane-jét }

@putBOB:
mov     al,2
out     dx,ax      { A következő plane írásra engedélyezése }
shl     ah,1      { A bitplane mutató a következő plane-re
                       áll }

mov     cx,16
push    di      { A képcímre még szükség lesz }

@putplane:
mov     al,ES:[di] { A latch feltöltése }
movsb
{ Az adat kivitele, OR kapcsolatba kerül a
  latch-csel! }

mov     al,ES:[di] { 2. bájt }
movsb
mov     al,ES:[di] { 3. bájt }
movsb
add     di,Lsize-3 { A képcím növelése, a következő sor }
loop    @putplane { Egy bitplane kirakása }
pop     di
test    ah,16
jz      @putBOB    { Visszaugrik, ha még van hátra bitplane }

```

A *@putplane* ciklus a BOB egy bitplane-jének kirakását végzi, míg a *@putBOB* ciklus lefutása után az egész BOB kint lesz az inaktív képen.

```

mov     ax,0f02h      { A következő BOB-maszok miatt szükséges
                       az összes }
out     dx,ax      { bitplane engedélyezése }

pop     cx
add     bx,BOBsize    { Veszi a következő BOB adatait }
loop    @main2        { Minden BOB-ot megjelenít }

```

A BOB-ok kirakása után — a már ismert módszerrel — a vertikális visszafutás alatt megcseréljük a két munkaképet, így a BOB-ok láthatóvá válnak.

```

{ Most kell megcserélni a két képet, az inaktívat láthatóvá kell
  tenni }

mov     dx,03d4h          { 3D4,0C,0D: képkezdőcím regiszter }
mov     al,0ch
mov     ah,byte [InacScr+1] { Cím HI bájt }
out     dx,ax
mov     al,0dh
mov     ah,byte [InacScr]   { Cím LO bájt }
out     dx,ax

mov     dx,03dah
@wait:  { Várakozás a vertikális visszafutásra }
in      al,dx
and     al,8
jz      @wait

        { Cserék }
xor     Inactive,0258h      { A szegmenscím cseréje }
xor     InacScr,2580h       { A képkezdőcím cseréje }
xor     Act,2               { A címtárkorrigáló adat cseréje }

pop     bp
pop     DS
End;
```

A BOB-ok mozgatására a korábban megismert két mozgató algoritmust alkalmazzuk.

```

{ Egy adott számú mozgásprogram megkeresése }
Function Prog(b:byte):word; Assembler;
  Asm
    ... az előzőekben már megismert függvény ... (BOB2.pas)
  End;

{ A BOB-ok mozgatása előre meghatározott mozgásprogram szerint }
Procedure MoveBOBS_A;
  Begin
    ... az előzőekben már megismert eljárás ... (BOB2.pas)
  End;

{ A BOBok mozgatása véletlenszerűen }
Procedure MoveBOBS_B;
  Begin
    ... az előzőekben már megismert eljárás ... (BOB1.pas)
  End;
```


A főprogramban inicializálnunk kell a címszámító táblázatokat és a képernyőket. Az inicializálás után a BOB-okat egy előre meghatározott program szerint végigvisszük a képen, majd a kép közepénél a mozgatót átveszi a véletlenszerűen mozgató algoritmus.

BEGIN

```
{ A 320x240_c16 mód bekapcsolása }
_320x240_c16;
```

```
{ Táblázat inicializálások }
```

```
for w:=0 to 15 do SadrTab[ w ] :=w*1536+ofs(Shapes);
for w:=0 to 15 do MaddrTab[ w ] :=w*384+ofs(Masks);
for w:=0 to 7 do XpixTab[ w ] :=w*48;
for w:=0 to 7 do XpixTab2[ w ] :=w*192;
for w:=0 to 479 do AddrTab[ w ] :=w*Lsize;
```

```
{ Az adatok beolvasása, eltolás és a maszk létrehozása }
Convert_Datas;
```

```
{ 1000 db véletlenszerű pont kirakása a képre: ez a háttér }
```

```
for i:=0 to 999 do Putpixel(random(320),random(240),1+random(15));
```

```
Screen_Init; { Képernyő inicializálások }
```

```
{ A BOB-ok adatainak feltöltése }
```

```
for i:=1 to BOBnum do
```

```
  with BOBS[ i ] do
```

```
    Begin
```

```
      X:=8+(pred(i) and 3)*16;
```

```
      Y:=8+(pred(i) shr 2)*16;
```

```
      S:=i-1;
```

```
      AdA:=0;
```

```
      AdB:=0;
```

```
      Prg:=0;
```

```
      Ppt:=Prog(Prg);
```

```
      Ir:=MovePrg[ Ppt ] ;
```

```
      Lp:=MovePrg[ Ppt+1 ] ;
```

```
      inc(Ppt);
```

```
    End;
```

```
{ Első mozgásciklus a programozott mozgítás szerint }
```

```
for w:=0 to 470 do
```

```
  Begin
```

```
    ShowBOBS;
```

```
    MoveBOBS_A;
```

```
    delay(5);
```

```
  End;
```

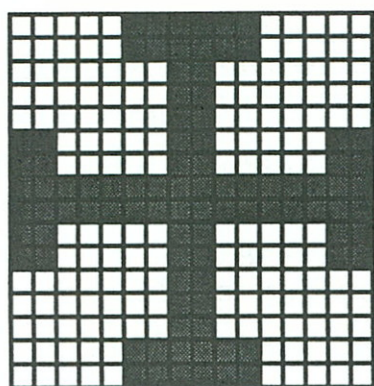
```
{ Második mozgásciklus a véletlenszerű mozgás szerint }  
repeat  
  ShowBOBS;  
  MoveBOBS_B;  
  delay(5);  
until keypressed; { Egy billentyű lenyomására kilép      }  
while keypressed do readkey; { A billentyűzetbuffer törlése }  
  
textmode(c80);  
END.
```

A program a *bob2.dat* adatokat tölti be, de a lemezen található *bob1.dat* is többszínű 16×16-os BOB-okat tartalmaz, így a program ezzel a fájlal is fut.

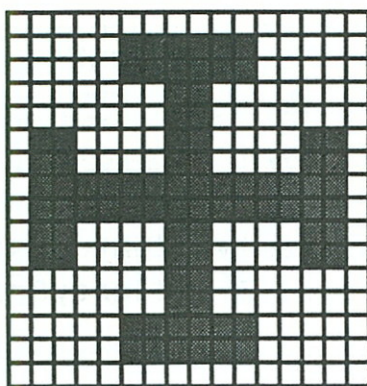
Ezzel a BOB-megjelenítővel lezárjuk a 16 színű BOB-programozást. A bemutatott négy program csupán oktató-ismertető jellegű volt, bizonyos rutinokat még lehet optimalizálni bennük. Ezért azt javasoljuk az Olvasónak, hogy kísérletezgesen bátran, próbáljon ésszerűsíteni és gyorsítani a programokon! A VGA kártya tulajdonságai még sok lehetőséget rejtenek magukban, elbizakodottság lenne tehát azt hinni, hogy mindent tudunk már a BOB-ok programozásáról. Ezek a demoprogramok csupán a vázát képezhetik valamely nagyobb feladat megoldásának. Hogy a további lehetőségekre most nem térünk ki, annak az az oka, hogy a 16 színű grafikus kép tulajdonképpen igencsak megnehezíti a dolgunkat, míg a 256 színű módok — noha nagyobb területet foglalnak, és a képszerkezetük is más — furcsa módon mégis alkalmasabbnak látszanak a BOB-ok megjelenítésére és mozgására.

7.10. Animáció, BOB mozgásfázisok

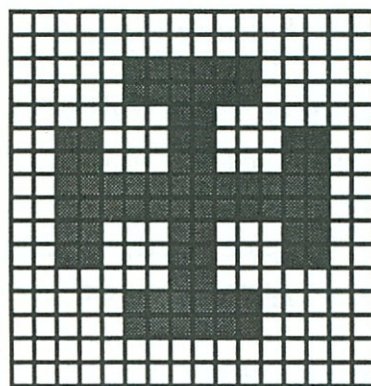
Az animációról eddig nem esett szó. A BOB-ok megjelenítésekor ez már jelentős változtatást nem igényel. A karakteres kép programozásának tárgyalásakor már érintettük az animáció alapjait, ott egy futószalagot mozgattunk animációs fázisokkal. A BOB-ok esetében is mozgásfázisokat kell létrehoznunk. Most itt az ismétlődő, önálló animációval fogunk foglalkozni. A kifejezés azt jelenti, hogy a mozgás periodikusan ismétli önmagát. Nézzük meg egy példa segítségével, hogy ez konkrétan mit is jelent!



1. animációs fázis



2. animációs fázis



3. animációs fázis

42. ábra Animációs fázisok

A fenti háromfázisú mozgás akkor lesz folyamatos és önmagába visszatérő, ha a fázisok az 1,2,3,2,1,2,3,2,1... sorrendben követik egymást. Az animáció annál élethűbb, minél kisebb az eltérés két fázis között, és minél több fázisból áll a mozgás. Mivel most csak az alapelv bemutatása a cél, ezért kevés fázissal dolgozunk. A BOB megjelenítésekor szükség van egy animációs korrigáló értékre, amely megadja, hogy hányadik animációs fázist (Shape-et) kell megjelenítenünk. Az **anim.dat** fájl négy darab háromfázisú Shape adatot tartalmaz, ezt fogjuk használni a programunkban. A program egyébként a **bob1.pas** módosított változata. A programban a *TimerIRQ* rutin végzi a BOB-ok animálását. Mivel a program nagy része azonos a **bob1**-gyel, ezért most csak ez utóbbi eljárást nézzük meg részletesebben:



ANIMATE.PAS

Az animációhoz a következő változókat kell deklarálnunk a programban:

```
Fspd      = 3;
Fnum      : byte = 0;
Fofs      : Array [ 0..3] of byte = (0, 1, 2, 1);
Fcor      : byte = 0;
Fctr      : byte = Fspd;
```

Az *Fspd* az animáció sebességét határozza meg. A \$1C megszakítás másodpercenként 18,2-szer fut le, de mi csak minden *Fspd*-edik megszakításkor váltunk fázist. Az animáció az *Fspd*=1 esetben a leggyorsabb, ekkor minden egyes időzítő megszakításkor történik fázisváltás.

Az *Fnum* valamennyi animáció esetén megnő, alsó két bitje szolgáltatja az indexet az *Fofs* tömbhöz, ahonnan kivesszük a megfelelő korrigáló értéket.

Az *Fofs* tartalmazza a fáziskorrigáló értékeket. A Shape adatokat úgy terveztük, hogy az animációs fázisok egymás után következzenek:

1. bob1.fázis, 1. bob.2.fázis, 1. bob.3.fázis, 2. bob.1.fázis...

Az *Fofs* tömbből származó értéket hozzáadjuk a BOB rekord *S* értékéhez, így kapjuk meg a következő fázist. Természetesen a generáláskor gondoskodni kell arról, hogy az *S* mező minden Shape legelső mozgásfázisára mutasson.

Az *Fcor*-t feltöltjük az *Fofs* táblázatból kivett értékkel, amelyet majd a BOB-megjelenítő *ShowBOBS* rutin hozzáad a Shape-számhoz.

Az *Fctr* számlálja a megszakításokat, ezt mindig az *Fspd* értékkel töltjük fel.

Procedure TimerIRQ; **Assembler;**

Asm

```
push DS
push ax
push bx          { A használt regisztereket tároljuk a STACK-ben }
```

```
mov ax,seg @data
mov DS,ax        { A DS-t ráállítjuk az eredeti adatszegmensre }
```

```
dec Fctr          { Az animációs lépésszámláló csökkentése }
jnz @endIRQ       { Kiugrik, ha még nem kell fázist váltani }
mov Fctr,Fspd     { Fázisváltáskor inicializálja a lépésszámlálót }
inc Fnum          { Növeli a fázismutatót (az alsó két bit kell) }
lea bx,Fofs       { Az eltolások táblázata }
mov al,Fnum       { A fázisszám az AL-be kerül }
and al,3          { Az alsó két bitet megtartja }
xlat              { Kiveszi a megfelelő fáziseltolást }
mov Fcor,al       { Feltölti a fáziskorrigáló változót }
```

```
@endIRQ:
pop bx            { Visszatölti az eredeti regiszterértékeket }
pop ax
pop DS
iret              { Visszatérés a megszakításból }
```

End;

A program futtatásakor a BOB-ok a már megismert módon mozognak a képen, és közben animációt is végeznek. A timermegszakítással végzett animációnak az az előnye, hogy a fázismozgás sebessége nem függ a futási időtől, vagyis a gép sebességétől.

7.11. Grafikus kép görgetése

A karakteres görgető (scroll) rutinokhoz hasonlóan itt is kétféle görgetés megvalósításáról beszélhetünk, az ablakon belüli és a teljes kép görgetéséről. A bitplane szervezés sajnos megnehezíti a dolgunkat, olyannyira, hogy például az ablakon belüli horizontális eltolást nem is nagyon tudjuk megvalósítani. A teljes kép mozgatása esetén azonban alkalmazhatjuk a karakteres képek programozásánál már megismert globális és ablakkép módszert.

7.11.1. Ablakon belüli görgetés

Egy adott ablakon belüli eltolás a színes bájtok mozgatását vagy a bitplane-ek külön-külön való kezelését jelenti. A színes bájtokat a VGA kártya másolhatja, elforgatni viszont nem tudja azokat. Ebből és a képernyő szerkezetéből adódik, hogy a jobbra-balra való eltolás igen nehéz, hiszen itt bitenként kell tologatni minden egyes színes bájtot. A fel-le mozgatásnál már más a helyzet, mivel egy vízszintes pixelsor lényegében színes bájtok egymásutánja, azaz itt egyszerű memóriamásolással pixeles elmozdulást tudunk létrehozni. A görgetés elvét a karakteres kép tárgyalásakor már megismertük, itt is ugyanazt kell tennünk: bájtonként egy sorral feljebb vagy lejjebb kell másolni az adatokat. Erre természetesen a **Mode 1** írás módot használjuk. Az alábbi *ScrollUp* rutin sok hasonlóságot mutat a karakteres *ScrollUp* rutinnal, de itt persze nem word adatokat, hanem bájtokat kell másolni:

*{Az X paraméterek bájtokat határoznak meg, ezért az értékük 0..79 közötti}
{lehet. Az Y paraméterek rastersorokat (pixelsorokat) határoznak meg}*

Procedure ScrollUp(X1,Y1,X2,Y2:word); **Assembler;**

Asm

push 0001

call WriteMode { **Mode 1** írás módba kell kapcsolni (lásd később) }

{ Most meg kell határozni az ablak bal felső sarkához tartozó képcímet }

mov ax,Lsize

mul Y1 { Hossz * Y }

add ax,X1 { Cím = hossz * Y + X }

mov di,ax { Az ES:DI lesz a célcím }

mov si,ax

add si,Lsize { Az ES:SI pedig a forrás (egy pixelsorral alatta) }

mov cx,X2

```

sub    cx,X1
inc    cx           { Az ablak szélessége bájtokban a CX-be kerül }
mov    dx,Y2
sub    dx,Y1       { Az ablak magassága pedig a DX-be }
mov    ES,segA000  { Az ES a grafikus képre mutat }

{ A következő ciklus egy pixellel elmozdítja az ablakot }
@cikl1:
push   cx
push   di
push   si
rep    seges movsb { A forrássort átviszi a célsorba
                  [ES:SI]->[ES:DI] }

pop    si
pop    di
pop    cx
add    si,Lsize    { Mindkét sormutatót növelni kell }
add    di,Lsize
dec    dx
jnz    @cikl1      { Minden sort feljebb kell másolni }
mov    al,ES:[38400] { Ez a kép mögötti első üres bájt
                  (fekete) }
        { A latch-be egy fekete színes bájt kerül }
rep    stosb       { Az utolsó sorba fekete bájtokat másol }
End;

```

A *ScrollLf* rutin egy ablak balra tolását valósítja meg, de mivel itt is színes bájtokat másolunk, ezért ez nyolcpixeles eltolást fog eredményezni. Az *X* paramétereknek itt is a 0..79 intervallumba kell esniük.

Procedure ScrollLf(X1,Y1,X2,Y2:word); **Assembler;**

```

Asm
push   0001
call   WriteMode   { Mode 1 írás mód }

mov    ax,Lsize    { Címszámítás a koordináták alapján }
mul    Y1
add    ax,X1
mov    di,ax       { Célcím a DI-be }
mov    si,ax
inc    si          { Forráscím az SI-be (eggyel több a célnál) }
mov    cx,X2
sub    cx,X1       { Vízszintes hossz a CX-be }
mov    dx,Y2
sub    dx,Y1       { Függőleges hossz a DX-be }
inc    dx
mov    ES,segA000  { Az ES-t a képre állítjuk }

@cikl1:

```



```

push  cx
push  si
push  di
rep  seges movsb  { Egy sor balra tolása egy bájtval          }
mov   al,ES:[ 38400]
stosb                { A sor végére egy fekete bájtot kell tenni  }
pop   di
pop   si
pop   cx
add   si,Lsize
add   di,Lsize      { A következő sorra áll mindkét mutató      }
dec   dx
jnz   @cikl1
End;

```

A *PlaneScroll* egy bitplane balra való eltolását végzi. Az ablak bal felső képbájtját az $X(0..79), Y(0..479)$ paraméterek határozzák meg, a H az ablak hossza bájtokban (nem lehet nulla), míg az M az ablak magassága pixelesen (ez sem lehet nulla). A W megadja az írni kívánt bitplane, az R pedig az olvasni kívánt bitplane számát.



SCROLL1.PAS

Procedure PlaneScroll(X, Y, H, M, W, R :integer); **Assembler;**

Asm

```

push  00
call  Writemode      { Mode 0 írás módban dolgozik          }
mov   dx,$3ce
mov   ax,R
out   dx,ax          { Read Plane select regiszter          }
mov   dx,$3c4
mov   ax,W            { Plane Write Enable regiszter        }
out   dx,ax
push  ds
mov   ds,segA000      { DS a képre mutat                  }

mov   ax,Lsize        { Képcím számítás                      }
mul   Y
add   ax,X
add   ax,H            { A balra tolást a végénél kell kezdeni }
dec   ax
mov   si,ax

mov   ax,Lsize
add   ax,H

mov   bx,M            { BX-be a magasság                      }
mov   dx,H            { DX-be a hosszúság                      }

```

```

mov    bx,M                { BX-be a magasság                }
mov    dx,H                { DX-be a hosszúság              }
clc
@c1:
mov    cx,dx
@c2:                { Egy sor eltolása egy pixellel          }
rcl    byte ptr [si],1
dec    si
loop   @c2
add    si,ax                { Növeli a sormutatót            }
dec    bx
jnz    @c1                { Minden sort eltol                }

pop    ds
End;

```

A lemez mellékleten található **scroll1.pas** programban megtalálhatók ezek a rutinok (és a segédrutinok is, mint pl. a *Writemode*). Amint azt az Olvasó is tapasztalhatja, ezek a rutinok nem túl gyorsak. Sajnos a bitplane szervezésű képeken az ablakon belüli görgetés csak kis ablakok esetén oldható meg. Nem volt még szó az ablak eltolásáról jobbra-balra pixelesen. Erre két módszer kínálkozik:

1. Az ablak minden bitplane-jét eltoljuk a fenti *PlaneScroll* rutinnal. Ez a módszer nem elég gyors, mert az első és az utolsó bitplane eltolása között még egészen kicsi ablakok esetén is történik legalább egy képfrissítés, ezért az eltolás villogással jár. Tapasztalataink szerint ezzel a módszerrel legfeljebb két bitplane, azaz négy szín tolható el villogásmentesen, de még ez is processzorajel függő.
2. Az ablak minden bitplane-jét kimásoljuk a képterület nem látható részére, ott elvégezzük az eltolást, majd mint színes bájtokat visszamásoljuk az eredményt. Ez a fajta módszer villogásmentes, lévén, hogy az eltolást egy nem látható területen végezzük, de ez is igen lassú.

Mindent összevetve meg kell állapítanunk, hogy a tizenhat színű képeken csak kis ablakokban tudunk görgetni, és még ez is nehézkes és körülményes.

7.11.2. Osztott képernyő és görgetés (scroll)

Amennyire nehéz a dolgunk az ablakon belüli görgetés esetében, annyival könnyebb az egész kép görgetésekor. Ugyanazon az elven, ahogyan a karakteres képnél létre tudtunk hozni globális és ablakképet, itt is meg tudjuk tenni mindezt, az egyetlen lényeges különbség, hogy a képen egyszerre látható memóriatartomány a grafikus képek esetében nagyságrendekkel nagyobb, mint a karakteres képeknél. Gondoljuk csak el, hogy egy 80×25-ös karakteres kép is mindössze 80*25*2, azaz 4000 bájt, míg egy

kis felbontású 320×200-as grafikus kép 8000 bájt akkor is, ha monokrom. Ha pedig ez a kép tizenhat színű, bitplane szervezésű, akkor a hossza 32 000 bájt. Most is — mint az előző fejezetekben már néhányszor — fel kell használnunk azt a tulajdonságot, hogy a képmemória a színes bájtok miatt a CPU szempontjából címtartományban csak negyedakkora, mint a valóságban. A létrehozható legnagyobb globális kép, amely 64 Kbájton elfér, 65 536 színes bájtot, tehát 8×65 536 pixelt tartalmazhat. Vannak olyan VGA kártyák, amelyek folyamatos képmemóriaként tudják kezelni a teljes 128K videoterületet (mint például a Trident), de erre nem minden kártya képes, ezért meg kell elégednünk a 64K összefüggő területtel. Ha például a globális kép 640 pixel széles — alapesetben is ekkora —, akkor a magassága legfeljebb 819 pixel lehet, ilyenkor a horizontális eltolást csak akkor lehet megvalósítani, ha az ablak 320 pixeles. Ha pedig a globális kép is csak 320 pixeles, akkor csak függőleges irányban tudunk scrollozni, legalábbis azzal a módszerrel, amelyet a karakteres képnél alkalmaztunk. Az ablakkép-pozicionáló eljárás, a *ShowPic* egyébként egyszerűsödik, mivel egyszerű címnöveléssel oldjuk meg a vertikális eltolást, hiszen itt nincs karakteren belüli függőleges eltolás. A vízszintes eltolást ugyanúgy a \$3C0, \$13 *horizontális eltolás* regiszterrel kell megvalósítanunk.

A következő példaprogram a scroll és az osztott képernyő használatát együtt mutatja be. Az osztott kép installálásához nem kell egyebet tenni, mint a \$3D4, \$18 *Line Compare* regiszterbe beírni a kettévágás rasztorsorát. A programban létrehozunk egy 640×480-as globális képet, aminek a felső negyven pixelét az osztott képernyő „B” részének fogjuk használni, míg a további részt az „A” képnek tartjuk fenn. Az ablakképet 320×240-es felbontásra állítjuk be. A program működése hasonló lesz a karakteres **scroll3.pas** programéhoz. (Ha az Olvasó már nem emlékszik erre, akkor azt javasoljuk, hogy tekintse át újból.)



SCROLL2.PAS

```
{ $G+}
```

```
Uses Crt,Crtplus,Grplus; {Felhasználjuk a saját UNIT-okat }
```

```
Const
```

```
    Lsize = 80; { A kép logikai hossza 80 bájt }
```

```
{ A CRTC regiszterek az osztott képernyős 320x240-es felbontáshoz }
```

```
_Data_240 : Array [ 0..$18] of byte=
```

```
    ($2d,$27,$28,$90,$2b,$80,$0b,$3e,
```

```
    $00,$80,$00,$00,$0c,$80,$00,$00,
```

```
    $ea,$8c,$df,$28,$00,$e7,$04,$e3,$90);
```

```
{ A Line Compare ($18) regiszterbe $190-et kell tölteni, hogy a  
    képet
```

```
{ a $190-edik (400-adik) sornál vágja ketté
```


Ez az adattömb tartalmazza a CRTC regiszterek értékeit, amelyek majd beállítják a 320×240-es ablakképet. Ez annyiban tér el a BOB-programozásnál használt beállítás-tól, hogy a képhosszt \$28-ra állítja, azaz 80 bájt lesz egy sorban. Ez azért szükséges, mert a globális képünk 640 pixel széles. A *Line Compare* regisztert is itt töltjük fel, így az ablakkép bekapcsolásakor az osztott képernyős üzemmód is automatikusan beindul.

```

Var
    i,j,k,l : integer; { Általános változók }

    Lad      : word;    { Az utolsó képcím tárolása }
    Px       : byte;    { A kép pixeles eltolásának tárolása }
    X,Y      : word;    { Az ablakkép koordinátái }

{ A kép átkapcsolása 320x240-es osztott üzemmódba }
Procedure _320x240_c16; Assembler;
Asm
    mov     dx,03c4h
    mov     ax,0901h      { A horizontális órajel átváltása 320-as
                           módba }

    out     dx,ax
    lea     si,_Data_240  { A CRTC adatok }
    mov     cx,019h      { $19 db regisztert kell átírni }
    mov     dx,03d4h
    mov     ax,0011h
    out     dx,ax        { A $11-es regiszter 7. bitjét törölni
                           kell }

    xor     al,al
@Set_regs:
    mov     ah,[ si]
    out     dx,ax
    inc     al
    inc     si
    loop    @Set_regs    { Feltölti a CRTC regisztereket }

    mov     dx,03dah      { Az alsó kép horizontális eltolását
                           tiltani kell }

    in      al,dx
    mov     dl,0c0h      { Ezt a $3C0, $10 regiszter 5. bitjével
                           tehetjük meg }

    mov     al,10h
    out     dx,al
    mov     al,21h      { Az 5. és a 0. bit 1-be }
    out     dx,al
    mov     al,20h      { A kép engedélyezése }
    out     dx,al
End;

```


A `_320x240_c16` lefutása után megjelenik az osztott képernyő, és a felbontás 320×240 lesz, amelynek a felső 200 pixele az „A” kép, az alsó 40 pixele pedig a „B” kép. Fontos, hogy a *Line Compare*-t nem 200-ra, hanem 400-ra kell állítani. Azt ugyanis ez a regiszter nem veszi figyelembe, hogy a rastersorok duplázottak-e vagy sem. A duplázást a `$3D4,$09 Maximum Scan Line` regiszter 7. bitjével állítjuk be, de a VGA kártyának ettől még ugyanúgy 480 sort kell felrajzolnia, ezért kell a *Line Compare*-be 400-at tölteni.

```
{ Az ablakkép pozicionálása a globális képen }
{ Az ablak mérete 320x200 pixel, a globális kép 640x440 pixeles }
Procedure ShowPic(X,Y:word); Assembler;
Asm

    { Először kiszámítja az új pozícióhoz tartozó címet és az X pixeles
    eltolást }

    add    Y,40          { A "B" kép miatt az Y koordinátát korrigálni
                           kell }
    mov    ax,Lsize      { Címszámítás }
    mul    Y
    mov    bx,X
    shr    bx,3
    add    bx,ax          { BX-be a pozícióhoz tartozó képcím kerül }
    mov    cx,X          { A pixeles eltolást az X alsó három bitje
                           adja }
    and    cl,7          { CL-be az X pixeles eltolás }

    { Elsőként a képcímet írja át, ha szükséges }

    cmp    bx,Lad
    jz     @l1
    mov    Lad,bx        { Csak akkor módosítja, ha változás volt }
    mov    dx,03d4h
    mov    al,0ch        { Cím HI regiszter $0c }
    mov    ah,bh
    out    dx,ax
    inc    al            { Cím LO regiszter $0d }
    mov    ah,bl
    out    dx,ax

    { A horizontális eltolás átírása }

@l1:          { Az X pixeles eltolás módosítása is csak
               akkor, }

    mov    al,cl
    cmp    al,Px        { ha változott a legutolsó értékhez képest }
```

```

    jz    @exit
    mov   Px,al
    mov   ah,al
    mov   dx,03dah    { Várakozás a vertikális visszafutásra }
@wait:
    in    al,dx
    and   al,8
    jz    @wait
    mov   dl,0c0h
    mov   al,13h      { Horizontális pixeles eltolás regiszter }
    out   dx,al
    mov   al,ah        { Az új pixeles eltolást az AH adja }
    out   dx,al
    mov   al,20h      { 5. bit = 1, a kép engedélyezése }
    out   dx,al
@exit:
    End;

```

A *ShowPic* eljárásnak nincs sok dolga. A megkapott *X,Y* koordinátákból kiszámítjuk a képcímet, valamint a horizontális eltolást egy bájtól belül. Ezek után a címet be kell tölteni a *\$3d4,\$0c,\$0d képkezdőcím* regiszterbe, de csak akkor, ha a cím módosult. Éppen ezért minden átíráskor az utolsó címet tároljuk a *LAD* változóban, és csak akkor írjuk újra a kezdőcímet, ha szükséges. Az *X* pixeles eltolással ugyanez a helyzet. A *Px* változó megadja az utoljára beállított horizontális eltolást. Ha volt változás, akkor ezt is újraírja. Mivel a *\$3C0, \$13 Attribute* regiszter, átírni csak visszafutás alatt lehet, különben zavart okoz a képen. Ezért szükséges a *@wait* ciklus, amely megvárja a következő vertikális visszafutást. A ciklus belsejében lévő **and al,8** utasítást kicserélhetjük **and al,9**-re, és ekkor nem csak a vertikális, hanem a horizontális visszafutás alatt is átírhatjuk a regisztert. Ez meggyorsítja az eljárást, de a lassúbb gépeken a kép jobb vagy bal szélén fekete villogó csíkozást okozhat. Ennek az az oka, hogy az *Attribute* regiszterek írásakor a képernyő tiltva van, ha tehát az átírás lassú, akkor a művelet „vége” belelóg a képbe, mert a horizontális visszafutás ideje nem elég a regiszter módosításához.

BEGIN

```

Screen_Init;    { Bekapcsoljuk a 640x480-as üzemmódot }

{ Kiírja a későbbi "B" képre kerülő szöveget }
Ink:=0;
Bbar(0,0,319,39,0);
Cbar(5,5,314,34,3);
_OutStrXY(21,5,'Osztott képernyő és pixeles scroll');
_OutStrXY(21,20,'      a 16 színű grafikus képen');
Ink:=14;

```



```

_OutStrXY(23,5,'Osztott képernyő és pixeles scroll');
_OutStrXY(23,20,'      a 16 színű grafikus képen');
Ink:=15;
_OutStrXY(400, 0,'Az ENTER lenyomása után a ');
_OutStrXY(400,13,'kurzorbillentyűkkel a kép ');
_OutStrXY(400,27,'mozgatható. Kilépés: ESC-re');

{ Véletlenszerűen telerajzolja az "A" képet }
Ink:=13;
Box(0,40,639,479);      { Keretvonal }

for i:=1 to 15 do      { Színes üres téglalapok }
  Begin
    ink:=i;
    box(i*8,40+i*8,640-i*8,480-i*8);
  End;

for i:=0 to 30 do      { Véletlenszerű, színes, kitöltött téglalapok }
  Begin
    k:=1+random(500);l:=random(300)+41;
    ink:=1+random(15);
    solidbar(1+k,1+l,k+5+random(125),l+5+random(150));
  End;

for i:=0 to 15 do      { Véletlenszerű, vastag keretű téglalapok }
  Begin
    k:=1+random(500);l:=random(300)+41;
    j:=random(3);
    Bbar(k,l,k+5+random(125),l+5+random(150),j);
  End;

for i:=0 to 15 do      { Véletlenszerű, vékony keretű téglalapok }
  Begin
    k:=1+random(500);l:=random(300)+41;
    j:=random(3);
    Cbar(1+k,1+l,k+5+random(125),l+5+random(150),j);
  End;

readln;                { Várakozás az ENTER lenyomására }

_320x240_c16;          { A 320x240-es oszott kép bekapcsolása }

InstallKeyIRQ;         { Aktiválja a saját billentyűmegszakítást }

```

```

{ A kurzorral most mozgatható a kép }

repeat
  if Ktb[ 72] and (y>0) then dec(y);
  if Ktb[ 77] and (x<320) then inc(x,2);
  if Ktb[ 75] and (x>0) then dec(x,2);
  if Ktb[ 80] and (y<239) then inc(y);
  Showpic(x,y);
  Delay(10);
until Ktb[ 1]; { Az ESC-re kilép a ciklusból }

UnInstallKeyIRQ; { Visszkapcsolja az eredeti
                  billentyűzetmegszakítást }

textmode(c80); { Kilépéskor visszaállítja a karakteres
                módot }

END.

```

A program a kép kirajzolása után várja az ENTER lenyomását. Az ENTER lenyomása után bekapcsoljuk az ablakképet és az osztott képernyős üzemmódot. A CrtPlus egységben lévő billentyűkezelő rutint felhasználva a kurzorral ezután mozgatható a kép. Az ESC hatására a program leáll.

Összefoglalva tehát, ha a grafikus képen mindenáron görgetni akarunk, akkor azt érdemesebb az egész képpel, illetve osztott képernyős módban tennünk, mert ezt a VGA kártya hardvere támogatja, míg az ablakon belüli görgetést legalább annyira megnehezíti.

7.12. Képmemória tárolása és mozgatása

7.12.1. Tárolás a heapben

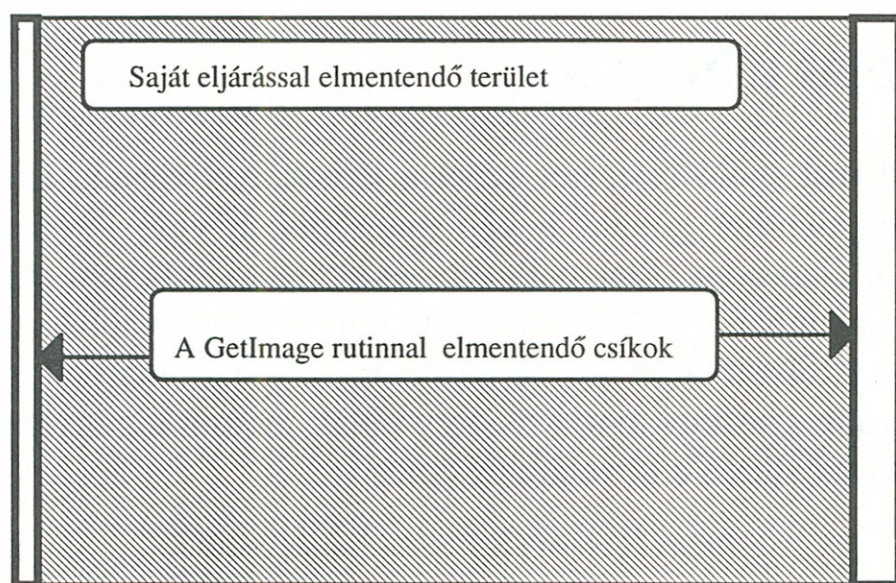
A 16 színű grafikáról szóló fejezetet ezzel a témakörrel zárjuk. Eddigi ismereteink alapján már nem jelenthet gondot a kép memóriában való tárolása vagy lemezre mentése. A Pascal *GetImage*, illetve *PutImage* eljárásai csaknem minden igényt kielégítenek. Sajnos azonban, mivel ezekkel a rutinokkal a heapben tároljuk el a kép valamely részét, a tárolható legnagyobb terület 65 528 bájt, márpedig ezt igen könnyű túllépni egy nagyobb grafikus mezővel. Egy ablak memóriaigényének meghatározására általánosan használható a következő képlet:

$$mem.igény = (X\ hossz * Y\ hossz) / 2$$

Ezzel a képlettel megtudjuk a pontos hosszúságot. Ha ez túllép a 64K-s határon, akkor a *GetImage* már nem használható. A méret meghatározására szolgáló *ImageSize*

függvény is hasonlóan számol, csak néhány adminisztrációs adat miatt ettől valamivel nagyobb területet határoz meg. Mit lehet tenni, ha a méret túlságosan nagy? Ilyenkor saját ablaktároló rutinokat kell írunk. A legegyszerűbb megoldás az lenne, ha pixelenként olvasnánk le a képet és úgy próbálnánk eltárolni. Programozástechnikailag ennek semmi akadálya, de ha valaki megír egy ilyen rutint, akkor azonnal látni fogja, hogy a lassúsága miatt használhatatlan.

A másik lehetséges megoldás, hogy az ilyen túlságosan nagy terület esetén bitplane-enként tároljuk el az ablakot. Ehhez persze egy helyett négy pointer és négy memóriaallokáció szükséges. Itt az ablakot természetesen a **Mode 0** olvasás móddal célszerű leolvasni. A rutin tehát kap egy téglalapot meghatározó méretet és négy pointert, amelyek a négy bitplane-t tároló memóriára mutatnak. Mielőtt továbbmennénk, máris felvetődik egy alapvető probléma. Nevezetesen, hogy a bitplane-t mi bájtonként akarjuk eltárolni, de a tárolandó ablak pixelesen van megadva, és ugyan miért esne pont bájthatárra a két széle. Azaz a jobb és a bal szélső adatokkal még baj lesz, ha a képet visszafelé másoljuk. Tehát, vagy valamilyen maszkolással elérjük, hogy a szélső bájtok helyesen tárolódjanak el, vagy az egészet figyelmen kívül hagyjuk. Persze bármelyik megoldást választjuk is, a visszamásolásnál ugyanebbe a problémába ütközünk. Ha korrekt módon szeretnénk végrehajtani a visszamásolást, akkor minden pixelsor első és utolsó bájtyát egy megfelelő maszk alapján össze kell montírozni a képen lévő bájtjal. Egyszóval: van bőven teendő. Mindezek miatt talán egyszerűbb egy áthidaló megoldást választani: az ablak két szélén használjuk a *GetImage* rutint, a belső rész kezelésére pedig írunk saját elmentő rutint. Így visszajutunk arra az esetre, hogy elég bájthatárokon elvégezni a tárolást, a két szélső csíkot ugyanis a *GetImage* is el tudja tárolni, hiszen egy ilyen csík legnagyobb mérete $7 \cdot 480/2$, azaz 1680 bájt. Az alábbi ábra alapján érthetőbbé válik a megoldás lényege.



43. ábra Nagy grafikus területek tárolása

Tehát nézzük a nagy ablakterületet elmentő eljárást! A bájthatáron történő lekérdezés miatt kell egy saját méretmeghatározó rutin is. Mivel a bitplane-ek természetesen azonos nagyságúak, ezért elég egynek a méretét meghatározni. A rutinoknak természetesen szükségük van néhány változóra és állandóra. A teljes program **grsave.pas** néven található meg a lemezmellékleten.



GRSAVE.PAS

{ \$G+}

Uses Crt, Grplus;**type**

ppt = ^par;

```
par = array[ 0..3] of pointer; { Egy négyelemű mutatótömb }
                                { Minden bitplane-hez egy-egy }
                                mutató }
```

const

Lsize = 80; { A kép szélessége bájtban }

{ A bitplane-ek kijelölése a \$3C4, \$02 regiszterrel }

Wpl: **Array**[0..3] **of** word=(\$0102,\$0202,\$0402,\$0802);

{ Ezekkel az állandókkal adjuk meg az ablak méretét }

wx1=0;

wy1=0;

wx2=79;

```
wy2=479; { Ezek az értékek az egész 640x480-as képet }
          definiálják }
```

var

i,j,k : integer; { Néhány általánosan használt változó }

w : word;

p : par; { Mutatótömb a bitplane-ekhez }

A nagy ablak méretének meghatározására készült a *PlaneSize* függvény. A koordinátákat bájtpozícióként kell megadni, tehát az X csak 0 és 79 közé eshet, míg az Y 0 és 479 közé. A függvény megadja egy bitplane hosszát bájtokban. Itt még a két sarokpont megadására nincs megkötés, de később már fontos lesz a sorrend.

Function PlaneSize(X1,Y1,X2,Y2:word):word; **Assembler;****Asm**

mov ax,X2

sub ax,X1

inc ax { A vízszintes méret az AX-be }

jns @t1

neg ax


```

@t1:
    mov    dx,Y2
    sub    dx,Y1
    inc    dx                { A függőleges méret a DX-be      }
    jns    @t2
    neg    dx
@t2:
    mul    dx                { A visszatérési érték az AX-ben  }
End;

```

A *GetPlane* eltárol egy bitplane-t. Az eljárásnak meg kell adni az ablak bal felső sarokbájtjának a címét (*Start*) a képmemóriában, valamint a szélességet bájtokban (*Xsize*) és a magasságot pixelesen (*Ysize*). A *Dest* egy típus nélküli változó, azaz mutató a tárolás helyére. A függvény önmagában természetesen nem működik jól, a későbbi *GetBIGImage* fogja majd használni.

```

Procedure GetPlane(Start,Xsize,Ysize:word; var Dest); Assembler;
Asm
    push DS
    mov  DS,SegA000 { A DS rámutat a forrásszegmensre      }
    mov  ax,Start
    les  di,Dest    { ES:DI mutat a memóriában a célcímre  }
    mov  dx,Ysize   { Az Y méretet a DX adja               }
    mov  bx,Xsize   { Az X méretet a BX adja               }
    cld
@saveplane:
    mov  cx,bx
    mov  si,ax
    rep movsb       { Egy bájtsor kimásolása a képről      }
    add  ax,Lsize   { A sormutató a következő sorra áll   }
    dec  dx
    jnz  @saveplane { Minden sort kimásol                  }
    pop  DS
End;

```

A *GetBIGImage* elvégzi a tulajdonképpeni mentést. A rutinnak meg kell adni az ablak méretét (az *X* méretet bájtathárként) és annak a pointertömbnek a címét, amelynek az elemei a négy lefoglalt heapterületre mutatnak. A rutinnak sorrendben kell megadni a koordinátákat, azaz az *X1,Y1* a bal felső sarkot, míg az *X2,Y2* a jobb alsó sarkot jelölje! A rutin a paraméterekből kiszámítja a méreteket és a kezdőcímet, majd a *GetPlane* négyszeri hívásával elmenti a képet. A *GetBIGImage* hívása előtt természetesen le kell foglalni a heapet.

```

Procedure GetBIGImage(X1,Y1,X2,Y2:word; p:ppt);
  var i,Start,Xsize,Ysize:word;
  Begin
    Start:=X1+Y1*Lsize; { A képkezdőcím kiszámítása }
    Xsize:=X2-X1+1;      { Az X hossz meghatározása }
    Ysize:=Y2-Y1+1;      { Az Y magasság meghatározása }

    for i:=0 to 3 do      { Négy bitplane-t kell elmenteni }
      Begin
        portw[ $3ce] :=i*256+4; { A Read Plane Select regiszterrel
                                kiválasztjuk }
                                { a következő bitplane-t az olvasáshoz }

        GetPlane(Start,Xsize,Ysize,p^[ i] ^); { Egy bitplane elmentése }
      End;
    End;

```

A következő *PutPlane* rutin már a kép visszatöltését végzi. Működése teljesen hasonló a *GetPlane* rutinéhoz, csak a forrás- és a célcím meg van cserélve, mivel itt a memóriából kell a képre másolni.

```

Procedure PutPlane(Start,Xsize,Ysize:word; var Source); Assembler;
  Asm
    push DS
    mov ES,SegA000
    mov ax,Start
    lds si,Source { DS:SI a memóriában a forrásra mutat }
    mov bx,Xsize
    mov dx,Ysize
    cld
  @loadplane:
    mov cx,bx
    mov di,ax
    rep movsb { Egy bájtsor visszamásolása a képre }
    add ax,Lsize { A sormutató a következő sorra áll }
    dec dx
    jnz @loadplane { Minden sort visszamásol a képre }
    pop DS
  End;

```

A *PutBIGImage* visszamásolja az eltárolt képet. Az eljárás hasonló a *GetBIGImage* rutinhoz, csak a forrás- és a célcím más, és persze a bitplane-eket most nem olvassuk, hanem írjuk, ezért a bitplane kiválasztását a \$3C4, \$02 Plane Write Enable regiszterrel végezzük el.


```

Procedure PutBIGImage(X1,Y1,X2,Y2:word; p:ppt);
  var i,Start,Xsize,Ysize:word;
  Begin
    Start:=X1+Y1*Lsize; { A képkezdőcím kiszámítása }
    Xsize:=X2-X1+1; { A szélesség meghatározása }
    Ysize:=Y2-Y1+1; { A magasság meghatározása }

    for i:=0 to 3 do { Négy bitplane-t kell másolni }
      Begin
        portw[$3c4] :=Wpl[i]; { A Write Plane Enable Select regiszterrel }
        { kiválasztjuk az írni kívánt bitplane-t }

        PutPlane(Start,Xsize,Ysize,p^[i]^); { Kiteszi a képre a
        bitplane-t }

      End;
    End;

```

A rutinok egy lehetséges használatát a **grsave.pas** program tartalmazza. A következő kiegészítésekre van még szükség:

```

  { A kép elmentése }
  w:=PlaneSize(wx1,wy1,wx2,wy2); { Meghatározzuk a méretet }
  for i:=0 to 3 do Getmem(p[i],w); { Lefoglaljuk a heapben a
  helyet }
  GetBIGImage(wx1,wy1,wx2,wy2, addr(p)); { Kimásoljuk a képet }

```

A fenti utasítások végrehajtása után a *wx1,wy1;wx2,wy2* paraméterekkel megadott ablak a heapbe kerül. A heap méretének ellenőrzéséről gondoskodni kell! Az eltárolt kép a következőképpen hívható vissza a képre:

```

  PutBIGImage(wx1,wy1,wx2,wy2, addr(p)); { Visszamásoljuk a képet}
  for i:=0 to 3 do Freemem(p[i],w); { Felszabadítjuk a memóriát }

```

Ezek után, ha a területet pixeles *X* határon is el akarjuk tárolni, akkor ezt még meg kell valósítanunk a normál *GetImage* és *PutImage* eljárásokkal, ahogyan azt a fejezet legelején leírtuk. Nem állítjuk, hogy ez a megoldás túl szép és jó, de be kell látnunk, hogy túlságosan nagy memóriaterületet kell kezelnünk, és sajnos a Turbo Pascal még nincs berendezkedve a 64K-nál nagyobb, összefüggő, dinamikus adatterületek kezelésére.

7.12.2. Tárolás a videomemóriában

A most következő képtárolási módszer lényege, hogy a képet a VGA kártya képmemóriájában tároljuk el. A módszer előnye a heapben való tárolással szemben az, hogy sokkal gyorsabb és jóval egyszerűbb, hátránya viszont, hogy viszonylag kevés terület

áll a rendelkezésünkre a tároláshoz, és hogy **a módszer nem minden VGA kártya esetén használható!** A következő feladatokat kell elvégeznünk:

- engedélyezni kell a 128K videmomemóriát, így a B000:0000 címtől 64K színes bájt-ként kezelhető terület áll a rendelkezésünkre a tároláshoz;
- meg kell írni az eltároló és a megjelenítő rutint. A rutinok **Mode 1** írás móddal fognak dolgozni.

A 128K videomemória bekapcsolása egyszerű, a *\$3CE*, *\$06 Memory Mode* regisztert kell 128K-ra beállítani:

Procedure Enable128K; **Assembler;**

Asm

```
mov    dx,03ceh
mov    ax,0106h
out    dx,ax      { 128K video-RAM engedélyezése }
```

End;

A most következő *StoreImage* eljárás elmenti a képterületet. A rutin statikusan a B000:0000 címre másolja a koordinátákkal megadott ablakot. A koordináták megadására itt is érvényes, hogy az *X*-ek bájtthátárokat jelölnek, és hogy a sorrend nem cserélhető fel ($X1 \leq X2$ és $Y1 \leq Y2$).

Procedure StoreImage (X1,Y1,X2,Y2:word); **Assembler;**

Asm

```
push    01
call    WriteMode { Mode 01 írás módot használunk }

push    DS
mov     ax,Lsize
mul     Y1
add     ax,X1
mov     si,ax      { SI-be a kezdőcím }
xor     di,di
mov     bx,X2
sub     bx,X1
inc     bx         { BX a vízszintes hossz }
mov     ax,Lsize
sub     ax,bx
mov     dx,Y2
sub     dx,Y1
inc     dx         { DX a függőleges hossz }
mov     ES,SegB000 { ES:DI a B000:0000 címre mutat }
mov     DS,SegA000 { DS:SI pedig az ablak első bájtjára }
```



```

@cikl:
    mov     cx,bx
    rep movsb      { Egy sor átmásolása }
    add     si,ax   { A sormutató növelése }
    dec     dx
    jnz     @cikl
    pop     DS

    push    00
    call    WriteMode { Visszaváltunk Mode 00 írásmódba }
End;

```

A *RestoreImage* eljárás hasonlóan működik, mint a *StoreImage*, csak a cél- és a forráscím fordított.

```

Procedure RestoreImage(X1,Y1,X2,Y2:word); Assembler;
Asm
    push    01
    call    WriteMode { Mode 01 írás mód }

    push    DS
    mov     ax,Lsize
    mul     Y1
    add     ax,X1
    mov     di,ax      { DI-be a kezdőcím }
    xor     si,si
    mov     bx,X2
    sub     bx,X1
    inc     bx          { BX a vízszintes hossz }
    mov     ax,Lsize
    sub     ax,bx
    mov     dx,Y2
    sub     dx,Y1
    inc     dx          { DX a függőleges hossz }
    mov     ES,SegA000 { ES:DI a célra mutat }
    mov     DS,SegB000 { DS:SI pedig a forrásra }

@cikl:
    mov     cx,bx
    rep movsb      { Egy sor átmásolása }
    add     di,ax
    dec     dx
    jnz     @cikl
    pop     DS

    push    00
    call    WriteMode { Vissza Mode 00 írás módba }
End;

```

Ezek a rutinok is a **grsave.pas** programban találhatóak meg. Ha az eljárások nem működnek, akkor ez azt jelenti, hogy az adott VGA kártya nem tudja képbájt típusként kezelni a B000:0000—B000:FFFF területet!

7.12.3. Tárolás a háttértárolón

Most azt nézzük meg, hogyan lehet egy képet a lemezre menteni és onnan visszatölteni. A két eljárás, a *SavePicture* és a *LoadPicture* elvi megoldása ugyanaz, mint a heapben való tárolásé, azzal a különbséggel, hogy a leolvasott bitplane-eket nem a heapben, hanem a lemezen tároljuk el, illetve onnan olvassuk vissza.

A koordinátákat az előzőekhez hasonlóan kell definiálni, a fájl nevét pedig a *Filename* paraméterrel adhatjuk meg. Vigyáznunk kell, mert az eljárás nem ellenőrzi, hogy van-e elegendő szabad terület a lemezen.

```
Procedure SavePicture(X1,Y1,X2,Y2:word;Filename:string);
  var i,j,Start,Xsize,Ysize:word;
      f:file;
Begin
  Start:=X1+Y1*Lsize; { A képkezdőcím kiszámítása }
  Xsize:=X2-X1+1;      { A szélesség meghatározása }
  Ysize:=Y2-Y1;        { A magasság meghatározása }
  Assign(f,Filename);
  Rewrite(f,1);
  for i:=0 to 3 do      { Négy bitplane-t kell kimenteni }
  Begin
    portw[ $3ce] :=i*256+4; { A Read Plane Select regiszterrel
                             kiválasztjuk
                             { a következő bitplane-t az olvasáshoz }
    for j:=0 to Ysize do { Soronként kimenti a bitplane-t
    BlockWrite(f,mem[ $a00:Start+j*Lsize],Xsize);
  End;
  Close(f);
End;
```

A *LoadPicture* betölti a kimentett képet:

```
Procedure LoadPicture(X1,Y1,X2,Y2:word;Filename:string);
  var i,j,Start,Xsize,Ysize:word;
      f:file;
Begin
  Start:=X1+Y1*Lsize; { A képkezdőcím kiszámítása }
  Xsize:=X2-X1+1;      { A szélesség meghatározása }
```



```

Ysize:=Y2-Y1;           { A magasság meghatározása }
Assign(f,Filename);
Reset(f,1);
  for i:=0 to 3 do       { Négy bitplane-t kell visszaolvasni }
  Begin
    portw[ $3c4 ]:=Wpl[ i ]; { A Write Plane Enable regiszterrel }
                                { kiválasztjuk az írni kívánt bitplane-t }

    for j:=0 to Ysize do { Soronként kirakja a bitplane-t }
      BlockRead(f,mem[ $a000:Start+j*Lsize],Xsize);
    End;
  Close(f);
End;

```

A tizenhat színű grafikus módokról szóló fejezet végére értünk. Reméljük, hogy az Olvasó hasznosítani tudja az itt megismert elveket és megoldásokat. A rutinok bővítése, a rendszerbe foglalás és az ésszerűsítés már az Olvasó dolga. Ezek a programok itt csak az alapok — mint ahogyan az egész könyv csak alap — valami komplexebb feladat megoldásához. A regiszterek kínálta lehetőségeket közel sem vettük végig, szerintünk ez még a kártyák gyártóinak is nehézséget okozna. Mindezek azt jelentik, hogy sok felfedezésre váró témakör lehet még, amelyről itt nem is beszéltünk. Ha valaki kíváncsi a kártya másfajta felhasználására, annak a következő megoldást javasoljuk: nyomkövető programokkal meg kell próbálni végigkövetni a VGA-BIOS rutinjait, és részletesen ki kell elemezni a regiszterek beállításait. A VGA kártya egyik sajátos tulajdonsága, hogy több regiszter együttes beállítása a várttól eltérő hatásokat produkál, ezért gondosan meg kell vizsgálni minden egyes IN és OUT utasítást, valamint az összes port valamennyi bitjét, hogy egy adott beállítás teljesen korrekt és főképp minden részletében érthető legyen.

8. A 256 színű grafikus mód programozása I. rész



AG256MODE

Elérkeztünk a 256 színű grafikához, amely a VGA kártyát tulajdonképpen megkülönbözteti elődjétől, az EGA kártyától. Ez a grafikus üzemmód — a BIOS-ban a \$13-as számú — aránylag kis felbontással, de sok színnel dolgozik, ezért inkább a játékprogramozók használják előszeretettel. Alapesetben a felbontás 320×200, amihez a képszerkezeteknél már ismertetett *bájt/pixel* képtárolási forma tartozik. A *bájt/pixel* képszerkezet igen előnyös a pont- és a vonalrajzolás szempontjából. Ahogyan a későbbiekben látni fogjuk, lényegesen egyszerűbb a programozása, mint a bitplane-eké. Nagy hátránya viszont, hogy csaknem a teljes grafikus címtartományt lefoglalja (64 000 bájt hosszú a kép), ezért nincs lehetőségünk sem a felbontás növelésére, sem több képernyőlap használatára. Ezért a VGA kártya készítői gondoskodtak egy olyan lehetőségről is, amellyel változtatni tudunk ezen a szerkezeti felépítésen. Ezáltal nagyobb felbontásba is kapcsolhatunk és több képlap használata is megoldható. Ez az úgynevezett *módosított bájt/pixel* formátum, amelynek a kezelése hasonló a bitplane-ek kezeléséhez, de mégis 256 színt tudunk használni. A VGA-BIOS nem támogatja ezt a lehetőséget, de a VGA kártya regisztereinek dokumentációjában szerepel. Ezért ez a fejezet tulajdonképpen két nagy részből fog állni: az első részben a *normál bájt/pixel* formátumú kép programozását ismerjük meg, míg a második részben a *módosított bájt/pixel* kép lehetőségeit és programozását mutatjuk be. A Pascalhoz ma már forgalmaznak egy **VGA256.BGI** grafikus interfészt is, de ez jogi okok miatt nem kerülhetett fel a lemez-mellékletre, ezért a legfontosabb grafikus rutinokat saját magunk készítjük majd el.

A fejezetben a következő problémákra térünk ki:

- pixelrajzolás és lekérdezés;
- vonalrajzolás;
- téglalapok rajzolása;
- karakterek megjelenítése, szövegkiírás a grafikus képre;
- grafikus területek tárolása;
- ablakon belüli görgetés (csak a normál képszerkezet esetében);
- BOB-ok programozása a 256 színű képen (képfrissítési módszer);
- osztott képernyő és görgetés (módosított képszerkezet);
- a felbontás változtatása (módosított képszerkezet);
- BOB-ok a kettősbuffer-technikával (módosított képszerkezet).

Ezeket a problémákat mindkét képszerkezet esetében tárgyaljuk, így kétszer írunk meg minden egyes rutint.

Először tehát foglalkozunk a *normál bájt/pixel szerkezetű képpel*! A lemezmellékleten a **GR256.PAS** (amely egy unit forrásprogramja) az ehhez a témához tartozó eljárásokat és függvényeket tartalmazza, ezért most ennek az egységnek a teljes leírásán keresztül, a BOB-programok kivételével, végignézzük a fenti témaköröket!

Az egység *interfész* része a következő:



GR256.PAS

```
Unit Gr256;
{$G+}
{$L char8x8.obj}
```

INTERFACE

Const

```
{ Képszegmens }
SegA000 : word = $a000;

{ Színdefiníciók a bekeretezett téglalapokhoz }
Cols : Array[ 0..11] of byte=(15,8,7,8,15,7,15,8,3,8,15,3);

Light: byte = 15;    { A bal és a felső keret színe           }
Dark  : byte = 08;    { A jobb és az alsó keret színe         }
Back  : byte = 07;    { A háttér színe                       }

Ink   : byte = 01;    { Tintaszín a karakteres kiíráshoz      }
Paper: byte = 00;    { Papírszín a karakteres kiíráshoz      }
Over  : byte = 01;    { A háttérmegőrzés/-felülírás kapcsolója }

Vspace : word =8;    { Betűtávolság az OutStrXY rutinhoz     }

Cadr : word = 0000; { A char8x8 képcíme                       }

{ A PutImage logikai műveleteinek állandói }
CopyPut    = 0;
NormalPut  = 0;
XorPut     = 1;
OrPut      = 2;
AndPut     = 3;
NotPut     = 4;
```

{ Az egység eljárásai és függvényei }

```

Procedure PutPixel (X,Y:word;Color:byte);
Function GetPixel (X,Y:word):byte;
Procedure Line (x1,y1,x2,y2:word;Color:byte);
Procedure SetRGBReg (regnum,Red,Green,Blue:byte);
Procedure Bar (x1,y1,x2,y2:word;Color:byte);
Procedure Box (x1,y1,x2,y2:word;Color:byte);
Procedure Bbar (x1,y1,x2,y2,ty:word);
Procedure Cbar (x1,y1,x2,y2,ty:word);
Procedure OutCharXY (X,Y:word;C:char);
Procedure OutStrXY (X,Y:word; S:string);
Procedure GetImage (X1,Y1,X2,Y2:word; var Dest);
Procedure PutImage (X,Y:word; var Source; Bop:word);
Function ImageSize (X1,Y1,X2,Y2:word):word;
Procedure ScrollUp (X1,Y1,X2,Y2:word);
Procedure ScrollDn (X1,Y1,X2,Y2:word);
Procedure ScrollLf (X1,Y1,X2,Y2:word);
Procedure ScrollRg (X1,Y1,X2,Y2:word);
Procedure Screen_Init;
Procedure ClearScreen;

```

A továbbiakban, az *implementation* részben részletesen végigkövetjük az eljárások működését, és mire az egység végére érünk, a fejezet elején leírt feladatok nagy részére már tudni is fogjuk a megoldást.

IMPLEMENTATION

Const

```

Lsize    = 320;    { A kép logikai hossza }

{ A vonalhúzó(LINE) algoritmus címnövelő táblázata }
Ntab : Array[ 0..11] of integer=(-321,-320,-319,000,
                                -001, 000, 001,000,
                                319, 320, 321,000);

```

Var

```

{ 16-bites regiszterek tárolására használatos változók }
r0,r1,r2,r3 : word;
r4,r5,r6,r7 : word;

{ 8x8-as karakterkészlet a char8x8 rutin számára }
Procedure Charset; External;

```

A változókat és az állandókat majd mindig a hozzájuk tartozó eljárásnál ismertetjük.

8.1. Pont kirajzolása és lekérdezése

8.1.1. Egy pixel kigyújtása

A *PutPixel* eljárás *Color* színűre állítja az *X,Y* pozíciójú képpontot. Az eljárás elsőként a koordinátaához tartozó képcímet számítja ki az alábbi képlettel:

$$\text{Képcím} = \text{Lsize}(\text{azaz } 320) * Y + X$$

A cím kiszámítása után már csak ki kell tenni a memóriacímre a *Color* értékű bájtot, és a pont megjelenik.

Procedure PutPixel (X,Y:word;Color:byte); **Assembler;**

Asm

```

mov  ax,Lsize          { Címszámítás, ES:DI mutat a megfelelő
                        képpontra                                }
mul   Y
add   ax,X
mov   di,ax
mov   ES,SegA000
mov   al,Color          { A szín az AL-be                        }
mov   ES:[di],al        { A képpont kigyújtása                  }
```

End;

Ha ezt az eljárást összehasonlítjuk a bitplane képszerkezetnél megismert *Pixel* eljárással, akkor látható, hogy ez a módszer lényegesen egyszerűbb, hiszen a címszámításkor nem kell bitpozíciót számolni, sem pedig **Mode 2** írás módot használni. A *normál bájt/pixel* formátumnál egyébként **Mode 0** olvasás és írás mód van beállítva, de számunkra ez érdektelen, mivel itt nincsenek bitplane-ek.

Mielőtt továbbmennénk, pontosítsuk a bitplane-vezérlő regiszterek és a *bájt/pixel* formátum kapcsolatát! A bitplane regiszterek itt is befolyásolják a képet, és itt is létezik a latch, csak mi sem a regiszterek adta lehetőséget, sem a latch-et nem használjuk. Ha próbaképpen például írás módot váltunk (\$3CE, \$05), vagy mondjuk tiltjuk valamelyik bitplane írását (\$3C4, \$02), és ezek után a képre írunk, akkor látni lehet, hogy a regiszterek bizony befolyásolják az adatforgalmat. Éppen ezért a *normál bájt/pixel* formátum esetén a regisztereknek azon csoportját nem érdemes módosítani, amelyek valamilyen kapcsolatban vannak a bitplane-ek írásával vagy olvasásával.

A képmemóriába írásnál nincs szükség maszkra és bitforgatásra, csak egyszerűen ki kell írni a *Color* színt a megadott képbájtba.

8.1.2. Egy képpont színének lekérdezése

A pont kigyújtásához hasonlóan ez a függvény is igen egyszerű. A képcím kiszámítása után egy egyszerű memóriaolvasással megkapjuk a pixel színét.

```
Function Getpixel(X,Y:word):byte; Assembler;
Asm
    mov ax,Lsize      { Címszámítás, ES:DI mutat a megfelelő
                        képpontra }
    mul Y
    add ax,X
    mov di,ax         { DI-ben van a cím }
    mov ES,SegA000
    mov al,ES:[di]    { A képpont betöltése az AL regiszterbe }
End;
```

8.2. Vonalrajzolás

A tizenhat színű kép programozásánál már megismertük a vonalrajzolás egyik egyszerű algoritmusát. A most következő *Line* eljárás az ott leírt algoritmusnak egy módosított assembly nyelvű változata lesz. A leglényegesebb módosítás az ott ismertett eljáráshoz képest — leszámítva, hogy az egy Pascal program volt, ez pedig egy Assembler rutin — az, hogy itt nem választjuk szét a vonalak fajtáit, hanem univerzálisan minden vonalat ugyanazzal a ciklussal fogunk kirajzolni. Ezt azért tehetjük meg, mert a 256 színű kép esetében feleslegessé válik mindenféle címszámítási algoritmus, amely az éppen következő pixel címének meghatározására szolgálna. Az állandóknál deklarált *Ntab* ugyanis tartalmazza a vonalhúzáskor lehetséges elmozdulásokat minden irányban, így a címszámítást egy egyszerű összeadás helyettesíti. Azt, hogy melyik irányba kell továbblépni, az *X* és az *Y* irányok egyértelműen megadják. Az algoritmus először kiszámítja az első pixel címét, és kiteszi a kezdőpontot. Ezek után az *X1,Y1* kezdő- és az *X2,Y2* végpont pozícióból meghatározza a következő értékeket:

<i>abs X=ABS(X2-X1)</i>	: abszolút X elmozdulás,
<i>abs Y=ABS(Y2-Y1)</i>	: abszolút Y elmozdulás,
<i>sgn X=SGN(X2-X1)</i>	: az X elmozdulás iránya átlós lépésnél,
<i>sgn Y=SGN(Y2-Y1)</i>	: az Y elmozdulás iránya átlós lépésnél.

Az *sgn X* és az *sgn Y* értéket (az *r2*-ben tároljuk) csak az átló irányú lépéshez határoztuk meg, értékük vagy +1, vagy -1, de semmiképpen sem nulla. Ez zavart okozhatna

a függőleges vagy a vízszintes vonalak esetén, de ezekben az esetekben a rutin nem használja ezeket az értékeket. Az *abs X* és az *abs Y* értékek két fontos információt hordoznak: egyrészt a kettő közül a kisebb megadja, hogy melyik lesz a főirány (a főirányt az *r1*-ben tároljuk és a *CX* a ciklusszámláló), másrészt a nem átló irányú lépéseknél a nagyobbhoz nulla elmozdulás tartozik (BH vagy BL kinullázása). Azt, hogy átlós vagy főirányú lépés következik-e, a növekmény $\text{INT}(H/2)$ -höz (az *AX*-ben van) való hozzáadása dönti el. A *H* az *abs X* és az *abs Y* közül a kisebb. Az előzetes számítások után a *@line_loop* ciklus rajzolja ki a pontokat. A ciklusban egy egyszerű összeadással döntjük el, hogy a következő lépés átló vagy főirányú-e. Ettől függően az *AX*-be betöltjük valamelyik elmozdulást. Átlós elmozdulás esetén az *r2* értéke kell, míg főirány esetén a *BX* adja az elmozdulást. Így az *AH* és az *AL* -1 , 0 vagy $+1$ értéket tartalmaz. Az *AX* megadja, hogy a *Htab* melyik elemére van szükség a következő pont címének kiszámításához:

AL(<i>abs X</i>)		→			
AH(<i>abs Y</i>)					
			-1	0	1
-1	↓	-321	-320	-319	?
0		-001	?	001	?
1		319	320	321	?

44. ábra A vonalhúzás elmozdulásai

A táblázat „?” elemei érdektelenek, mert ezek a helyek redundánsak, azaz nem fordulhat elő, hogy az indexértékek oda mutassanak. A vízszintes irányban csak azért helyeztünk el négy elemet, hogy a számítás könnyebb legyen. A táblázatból kivett értéket ezt követően egyszerűen hozzáadjuk az utoljára kitett pont képcíméhez (DI tartalmazza a címet), és kirajzoljuk a pontot. A vonal akkor készül el teljesen, ha a *CX* eléri a nullát.

Procedure Line(*x1,y1,x2,y2:word;Color:byte*); **Assembler;**

Asm

```

mov  ES,SegA000      { Az ES-t ráállítjuk a képre }
{ Az első lépésben kiszámítjuk a kezdőpont címét,
  és kirajzoljuk azt }
mov  ax,Lsize        { Címszámítás (cím=320*Y+X) }
mul  y1
add  ax,x1
```

```

mov  di,ax          { A DI a kezdőponthoz tartozó címet
                    tartalmazza }

mov  al,Color
mov  ES:[di],al      { A kezdőpont kirajzolása a képre }

{ Most következik az abs X, abs Y, sgn X és sgn Y előállítása }

mov  bx,0101h        { sgn X (BL) és sgn Y (BH), mindkettő +1 }
mov  dx,X2
sub  dx,X1           { DX = X2-X1 }
jnc  @t1             { Ha a kivonás alulcsordul, akkor negálni
                    kell }
neg  dx              { DX = abs X (ahol X=X2-X1) }
mov  bl,255          { BL = sgn X }
@t1:
mov  si,Y2
sub  si,Y1           { SI = Y2-Y1 }
jnc  @t2             { Ha a kivonás alulcsordul, akkor negálni
                    kell }
neg  si              { SI = abs Y (ahol Y=Y2-Y1) }
mov  bh,255          { BH = sgn Y }
@t2:
mov  cx,si           { CX = abs Y }
mov  r2,bx           { r2 = sgn X sgn Y tárolása későbbi
                    használatra }

{ A főirány meghatározása az abs X >= abs Y relációval }

cmp  dx,si           { Eldönti, hogy melyik lesz a főirány (X/Y) }
jnc  @x_ge_y         { Ugrás, ha abs X >= abs Y }

{ X főirányú a vonal }
mov  r1,dx           { r1 = abs X }
xor  bl,bl           { sgn X = 0, tehát az X növekményes ferde
                    vonal }

jmp  @t3

@x_ge_y:
test dx,dx           { Akkor lép ki, ha abs X=abs Y=0
                    (csak egy pont) }

jz   @exit
{ Y főirányú a vonal }
mov  r1,si           { r1 = abs Y }
mov  cx,dx           { cx = abs X }
xor  bh,bh           { sgn Y = 0, tehát az Y növekményes ferde
                    vonal }

```



```

@t3:
    mov    r3,cx          { CX darab pontot kell kirakni          }
    mov    ax,cx
    shr    ax,1           { Az AX-be kerül az int(CX/2) érték    }
    { A @line-loop ciklus a vonal pontjait rajzolja fel          }

@line_loop:
    add    ax,r1          { A növekmény hozzáadása az AX-hez      }
    jc     @diag          { Ha túlcsordul, akkor egy átló irányú  }
                        lépés következik                          }

    cmp    ax,r3
    jc     @vhl           { Ha kisebb az r3-nál, akkor V vagy H    }
                        lépés (főirány)                          }

@diag:          { Átlós lépés                                    }
    sub    ax,r3          { Az összeadás korrigálása              }
    mov    r4,ax          { Az AX tárolása későbbre              }
    mov    ax,r2          { Az eredeti sgn X és sgn Y érték kell    }
    jmp    @nextplot      { Ugrás a pontrajzolásra                }

@vhl:          { Vízszintes vagy horizontális főirányú lépés    }
    }
    mov    r4,ax          { Az AX tárolása későbbre              }
    mov    ax,bx          { A számított sgn X és sgn Y kell        }
                        (valamelyik nulla)                        }

@nextplot:      { Pontrajzolás                                    }
    mov    r5,bx          { A BX regiszter tárolása                }
    { Most ki kell számítani az Ntab indexét (AX)                }
    inc    al
    shl    al,1           { AL = (sgn X+1)*2, ez 0,2,4 érték lehet }
    inc    ah
    shl    ah,3           { AH = (sgn Y+1)*8, ez 0,8,16 érték lehet }
    add    al,ah
    xor    ah,ah          { Az Ntab indexe most az AX-ben van      }
    lea    bx,Ntab        { Az Ntab kezdőcíme a BX-be              }
    add    bx,ax          { A BX most az Ntab megfelelő elemére mutat }
    add    di,[ bx]       { A képcím növelése, elmozdulás a megfelelő }
                        irányba                                    }

    mov    al,Color
    mov    ES:[ di ],al   { A pont kirajzolása                    }

    mov    ax,r4          { A tárolt értékek visszahívása        }
    mov    bx,r5
    loop   @line_loop     { Vissza, ha van még kirajzolandó képpont }

@exit:
    End;

```

Az általános *Line* a képernyő szerkezetének köszönhetően meglepően gyors.

8.3. RGB beállítás

A *SetRGBreg* eljárást már ismerjük a paletta programozásából. Csupán azért szerepel ebben az egységben, hogy itt is legyen lehetőségünk a paletta módosítására. Az eljárás a *regnum* számú RGB regisztert beállítja a megadott *Red*, *Green* és *Blue* komponensekre.

Procedure SetRGBreg (regnum, Red, Green, Blue:byte); **Assembler;**

Asm

```

mov  dx,03c8h      { RGB írás címregiszter }
mov  al,regnum
out  dx,al         { A regiszterszám megadása }
inc  dx            { 03C9h az RGB adatregiszter }
mov  al,Red        { Piros komponens }
out  dx,al
mov  al,Green      { Zöld komponens }
out  dx,al
mov  al,Blue       { Kék komponens }
out  dx,al

```

End;

8.4. Téglalapok rajzolása

8.4.1. Kitöltött téglalap

A *BAR* eljárás kitöltött téglalapot rajzol. A koordináták két tetszőleges átlós sarokpontot adhatnak meg. Az eljárás először a koordináták alapján meghatározza a bal felső sarok címét, valamint az *X* és *Y* hosszúságot. Ezután *Y* darab *X* hosszúságú vízszintes vonal kirajzolásával létrehozza a téglalapot.

Procedure Bar (x1,y1,x2,y2:word;Color:byte); **Assembler;**

Asm

```

mov  ES,SegA000
cld

```

{ Az *X1* és *X2* közül kiválasztja a kisebbet, és meghatározza az *X* hosszúságot }

```

mov  ax,X1          { Az X koordináták vizsgálata }
mov  cx,X2
cmp  ax,cx          { Összehasonlítás }

```



```

    jc    @t1
    xchg  ax,cx          { Csere, ha szükséges }
    mov   X1,ax          { X1-be kerül a kisebb érték }
@t1:
    sub   cx,ax          { Kivonja a nagyobból a kisebbet, ez lesz
                          a hosszúság }
    inc   cx             { A CX-ben a vízszintes pixeles hosszúság
                          található }
    mov   di,ax          { Az X1-et a DI-be is beteszi }

    mov   si,Y2          { Az Y koordináták vizsgálata }
    mov   ax,Y1
    cmp   ax,si          { Összehasonlítás }
    jc    @t2
    xchg  ax,si          { Csere, ha szükséges }
    mov   Y1,ax          { Y1-be kerül a kisebb érték }
@t2:
    sub   si,ax          { Kivonja a nagyobból a kisebbet, ez lesz
                          a hosszúság }
    inc   si             { Az SI-ben a függőleges pixeles hosszúság
                          található }

    { Most AX=Y1 és DI=X1 }

    mov   bx,Lsize       { Címszámítás, a bal felső sarokhoz
                          tartozó képcím }
    mul   bx
    add   di,ax          { Az ES:DI a bal felső sarokpontra mutat }

    mov   al,Color       { A téglalap színe }

    sub   bx,cx          { A BX=Lsize-vízsz. méret (ez a
                          címnöveléshez kell) }
    mov   dx,cx          { A DX adja majd a vízszintes hosszúságot }

    { A téglalapot vízszintesen felrajzolt sorokból állítjuk össze }
@cikl1:
    mov   cx,dx          { A hossz betöltése a CX-be }
    rep  stosb           { Egy sor kirajzolása a képre }
    add   di,bx          { A DI-t a következő sorra állítja }
    dec   si
    jnz   @cikl1         { Minden sort kirajzol }
End;

```

8.4.2. Üres téglalap

Az üres téglalap kirajzolásakor az előbbi esethez hasonlóan meg kell határozni a bal felső sarok címét és a hosszúságokat. Ez után három különálló ciklus következik. Először felrajzoljuk a felső vízszintes vonalat. Utána a két függőleges vonal kirajzolása következik, majd utoljára az alsó vízszintes vonal.

Procedure Box(x1,y1,x2,y2:word;Color:byte); **Assembler;**

Asm

```

    mov     ES,SegA000
    cld
    mov     ax,X1                { Az X koordináták vizsgálata                }
    mov     cx,X2                {                                           }
    cmp     ax,cx                { Összehasonlítja a két X koordinátát    }
    jc      @t1
    xchg    ax,cx                { Csere, ha szükséges                    }
    mov     X1,ax                { Az X1-be a kisebb érték kerül          }
@t1:
    sub     cx,ax                { Kivonja a nagyobból a kisebbet          }
    inc     cx                    { A CX-ben a vízszintes pixeles          }
                                { hosszúság található                      }
    mov     di,ax                { DI-be teszi az X1-et                    }

    mov     si,Y2                { Az Y koordináták vizsgálata                }
    mov     ax,Y1                {                                           }
    cmp     ax,si                { Összehasonlítás                          }
    jc      @t2
    xchg    ax,si                { Csere, ha szükséges                    }
    mov     Y1,ax                { Az Y1-be a kisebb érték kerül          }
@t2:
    sub     si,ax                { Kivonja a nagyobból a kisebbet          }
    inc     si                    { Az SI-ben a függőleges pixeles          }
                                { hosszúság található                      }

    mov     bx,Lsize             { Címszámítás, a bal felső sarokhoz      }
                                { tartozó képcím                            }

    mul     bx
    add     di,ax                { Az ES:DI a bal felső sarokpontra      }
                                { mutat                                    }

    mov     al,Color             { A téglalap színe                          }

    mov     dx,cx                { A DX adja majd a vízszintes            }
                                { hosszúságot                             }

    mov     bx,di                { A címet tároljuk későbbre                }
```



```

{ Az első ciklus kirajzolja a felső vízszintes vonalat          }
rep stosb                { Ez a felső vízszintes vonal          }

dec  di                  { Csökkentjük a címet, az most az utolsó
                        pontra mutat                            }
mov  cx,si               { CX-be teszi a függőleges magasságot  }
dec  cx
jz   @exit               { Kilép, ha nincs magassága           }

{ A második ciklus a két függőleges vonalat rajzolja fel      }
{ A BX a bal felső sarok címét tartalmazza                     }
{ A DI a jobb felső sarok címét tartalmazza                    }
{ A CX a magasságot adja                                       }
{ Az AL-ben a szín található                                   }
@cikl:
mov  ES:[bx],al          { A bal oldali függőleges vonal       }
mov  ES:[di],al          { A jobb oldali függőleges vonal       }
add  bx,Lsize            { A címeket növeli                     }
add  di,Lsize
loop @cikl

{ A harmadik ciklus az alsó vízszintes vonalat rajzolja ki     }
mov  di,bx               { DI-be a bal alsó sarok címe         }
mov  cx,dx               { CX-be a vízszintes hosszúság        }
rep  stosb               { Ez az alsó vízszintes vonal         }
@exit:
End;

```

Látható, hogy a téglalapok rajzolása is leegyszerűsödik a bitplane szervezésű képeken való rajzoláshoz képest. A téglalapok rajzolása egyszerű memóriefeltöltéssé alakul, némi számítással bonyolítva. A *BAR* és a *BOX* esetében a koordináták két átlós pontra vonatkoztak, és nem volt lényeges a sorrendjük. A most következő keretezett téglalapok esetében ez már nem így lesz.

8.4.3. Vastagon keretezett téglalap

A *BBAR* eljárás az *X1,Y1* (bal felső) és az *X2,Y2* (jobb alsó) pontokkal definiált, vastagon keretezett téglalapot rajzolja ki a megadott *Ty* típussal. A típus alapesetben 0 és 3 között lehet, a *Cols* tömb adja meg a típusokhoz tartozó színeket. A tömbben három elem tartozik egy típushoz, amelyek sorban megadják a színeket:

Light: a bal és a felső (világos) oldalak színe,
Dark: a jobb és az alsó (sötét) oldalak színe,
Back: a téglalap alapszíne.

A *Cols* tömb bővítésével további típusokat lehet létrehozni, de ekkor a rutinban is módosítani kell a felső határt. Alapesetben ugyanis a $Ty \geq 4$ értéknél az eljárás az előző típussal beállított színeket veszi. A rutin működése egyszerű, a már megírt *BAR* és *Line* eljárások meghívásával állítjuk elő a keretezett téglalapot.

Procedure Bbar(x1,y1,x2,y2,ty:word); **Assembler;**

Asm

```

    cmp    Ty,4                { Nincs típusdefiníció, ha a Ty>=4      }
    { Ezt kell módosítani, ha további típusokat definiálunk        }
    jnc    @nodeftype
                                { Típusdefiníció, a színek feltöltése }

    mov    si,Ty
    add    si,si                { SI = Ty*2                            }
    add    si,Ty                { SI = Ty*2 + SI, azaz 3*Ty            }
    add    si,offset Cols       { Hozzáadjuk a tömb kezdőcímét        }

    { Most az SI mutat a megfelelő típushoz rendelt színekre      }
    mov    al,[si]              { [SI+0] a Light szín                 }
    mov    light,al
    mov    al,[si+2]            { [SI+2] a Back szín                 }
    mov    back,al
    mov    al,[si+1]            { [SI+1] a Dark szín                 }
    mov    dark,al

@nodeftype:
    { Elsőként a BAR meghívásával kirajzoljuk az alap téglalapot }
    push  X1
    push  Y1
    push  X2
    push  Y2                    { Koordináták                          }
    mov    al,Back              { Az alapszín                          }
    xor    ah,ah
    push  ax
    call  BAR                    { A háttértéglalap kirajzolása        }
    { A BAR-ral most az alsó sötét keretet rajzoljuk ki            }
    push  X2
    push  Y2
    push  X1
    mov    ax,Y2
    dec    ax
    push  ax
    mov    al,Dark
    xor    ah,ah
    push  ax
    call  BAR                    { Az alsó sötét keret kirajzolása    }

```



```

{ A jobb szélső sötét keretet is a BAR rajzolja ki }
push X2
push Y2
mov ax,X2
dec ax
push ax
push Y1
mov al,Dark
xor ah,ah
push ax
call BAR { A jobb szélső sötét keret
          kirajzolása }

{ A világos kereteket a Line eljárás rajzolja ki }
push X1
push Y1
push X2
push Y1
mov al,Light
xor ah,ah
push ax
call LINE { A világos keret fent, első vonal }
push X1
push Y1
push X1
push Y2
mov al,Light
xor ah,ah
push ax
call LINE { A világos keret bal szélén, első
          vonal }

inc Y1
dec X2
push X1
push Y1
push X2
push Y1
mov al,Light
xor ah,ah
push ax
call LINE { A világos keret fent, második vonal}
inc X1
dec Y2
push X1
push Y1
push X1
push Y2
mov al,Light

```

```

xor  ah,ah
push ax
call LINE                      { A világos keret bal szélén,
                                második vonal                      }

End;
```

8.4.4. Vékonyan keretezett téglalap

A *CBAR* ugyanazon az elven rajzol, mint a *BBAR*, de a keretvonalak itt csak egy pixel szélesek. A hasonlóság miatt itt ezt most nem részletezzük.

A keretezett téglalapok felrajzolása tehát nem ördögösség, hiszen csak a *Line* és a *BAR* hívogatásával állítjuk elő a rajzot. A rutinokban az assembly betétből Pascal rutinokat hívunk és ezeket paraméterezzük. Látható, hogy a vermet a deklarációkban megadott sorrendben kell feltölteni, és hogy a bájt méretű adatokat is szó méretű paraméterként kell átadni.

8.5. Karakterek kiírása a 256 színű képre

A most következő három eljárással 8×8-as karaktereket tudunk kiírni a 256 színű képre. A képszerkezet egyedül talán a karakterkiírásnak nem kedvez, de azért ez is megoldható, ha nem is éppen a leggyorsabban. A karaktereket ugyanis itt bitenként kell kinyomtatni a képre, minden egyes pixelt külön-külön. Ez 64 bájt adatmozgatást jelent karakterenként, amit még a tintaszín/papírszín vizsgálattal is kombinálni kell, ezért a karakter kiírása tovább tart, mint a bitplane szervezésű képeken. Ezzel szemben előny az, hogy a karaktereket nem kell bitenként elforgatni ahhoz, hogy pixelháttra tudjuk kirakni azokat. A most következő *char8x8* rutin az AL kódú karaktert nyomtatja ki, a későbbi rutinokból ezt fogjuk majd meghívni. A karakterkészletet a *char8x8.dat* fájl tartalmazza, ennek object fájl (*char8x8.obj*) változatát az egység kódszegmenséhez szerkesztve tároljuk, és *charset* néven tudunk hivatkozni rá. A konvertálást a BINOBJ Pascal segédprogrammal végeztük el.

Az eljárás bemenő paraméterei a következők:

- Ink:** ez adja meg az írás színét (1-es bitek a karakterboxban).
- Paper:** ez adja meg a papír színét (0-s bitek a karakterboxban).
- Over:** a háttér/papír kapcsolást végzi el. Az *Over=0* esetben a karakterbox 0 bithelyeire a Paper szín kerül, míg az *Over<>0* esetben a háttér változatlan marad a karakteren belül.
- Cadr:** a kiírás pozícióját, a képcímet adja meg. A cím a karakter bal felső sarokpontjának címét jelenti.

Felhasználjuk még a *Vspace* változót a képcím növelésére, azaz ez a változó adja meg két karakter bal felső sarkainak vízszintes távolságát pixelesen. Alapesetben ez az érték nyolc, de megváltoztatható, ezáltal vízszintesen sűríteni vagy ritkítani tudjuk a betűket.

Procedure Char8x8; Assembler;

Asm

```

    cld

    { Első lépésként kiszámítjuk a karakter kezdőcímét }

    xor    ah,ah
    shl    ax,3                { A kód megszorzása 8-cal }
    lea    si,Charset          { A karakterkészlet kezdőcíme }
    add    si,ax               { A CS:SI most a megfelelő karakterre mutat }
    mov    di,Cadr             { A DI-be tesszük a képcímet, az ES:DI a
                                képre mutat }
    cmp    Over,0              { A háttérre rajzolást az Over határozza meg }
    mov    ch,8                { A karakter 8 pixel magas, azaz 8 bájtból
                                áll }
    jz     @paper              { Over = 0 esetben a Paper színt viszi a
                                háttérre }

    { Over<>0, azaz a karakter most nem írja felül a hátteret }

    mov    ah,Ink              { Tintasín az AH-ba }
@c1:
    segcs  lodsb               { A karakterminta következő bájtja }
    mov    cl,8                { 8 bitből áll vízszintesen }
@c2:
    shl    al,1                { A bit kimegy a Carry-be }
    jnc    @t1                 { Ugrás, ha a bit nulla, ekkor nem tesz ki
                                semmit }
    mov    ES:[di],ah          { Kirak egy pontot, ha kell }
@t1:
    inc    di                  { A képcím növelése }
    dec    cl                  { A bitszám csökkentése }
    jnz    @c2                 { Egy bájt valamennyi bitjét megnézi }
    add    di,Lsize-8          { A DI a következő sorra áll rá }
    dec    ch                  { A bájt szám csökkentése }
    jnz    @c1                 { Kirajzolja a karakter 8 sorát }
    jmp    @exit               { Kilépés, ha a karakter teljesen kint van }

    { Over=0, azaz a háttérre most a Paper szín kerül }

```

```

@paper:
    segcs lodsb          { Beolvassa a karakterminta következő
                          bájtját }
    mov  cl,8            { 8 bitből áll vízszintesen }
@c3:
    shl  al,1            { A bit kimegy a Carry-be }
    jnc  @t2
    mov  ah,Ink          { Ink színű, ha a Carry=1 }
    mov  ES:[di],ah
    jmp  @t3
@t2:
    mov  ah,Paper        { Paper színű, ha a Carry=0 }
    mov  ES:[di],ah
@t3:
    inc  di              { A képcím növelése }
    dec  cl              { A bitszámláló csökkentése }
    jnz  @c3             { Minden bitet megvizsgál }
    add  di,Lsize-8      { A képcím növelése, a DI a következő sorra
                          mutat }
    dec  ch              { A bájt szám csökkentése }
    jnz  @paper          { Minden sort kirajzol }
@exit:
    mov  ax,Vspace       { Kilépés előtt a címet megnöveli a
                          betűtávolsággal, }
    add  Cadr,ax          { A Cadr így a következő kiírási pozíciót
                          adja }

End;

```

A *char8x8* önmagában nem működik, mert az ES regiszter nincs beállítva benne, ezért ezt az eljárást más rutinokból kell hívni.

8.5.1. Egy karakter kiírása

Az *OutCharXY* egy karaktert ír ki a megadott *X*, *Y* pozícióba, az *Ink*, a *Paper* és az *Over* változók figyelembevételével. Az eljárás az előbbi *char8x8* rutint hívja meg.

Procedure OutCharXY(*X*,*Y*:word;*C*:char); **Assembler;**

Asm

```

    mov  ax,Lsize        { Először kiszámítjuk a képcímet az X,Y
                          alapján }

    mul  Y
    add  ax,X
    mov  Cadr,ax         { A kiszámított címet a Cadr-ba teszi }
    mov  al,C            { A karakter kódja az AL-be kerül }
    mov  ES,SegA000      { Az ES a képre mutat }
    call Char8x8         { Kiírja a karaktert }

```

End;

8.5.2. Egy sztring kiírása

Az *OutStrXY* eljárás sztringeket ír ki. A pozíciót az *X,Y* adja meg, a színeket az előző esethez hasonlóan az *Ink*, *Paper* és *Over* változók határozzák meg. A sztring átadása érték szerint történik, de a cím szerinti átadás gyorsabb, ezért ha valaki előre definiált, az adatszegmensben elhelyezett szövegkonstansokat akar kiírni, akkor a **var** szóval érdemes a paraméterátadást cím szerintivé tenni.

Procedure OutStrXY(*X,Y:word; S:string*); **Assembler;**

Asm

```

    mov  ES,SegA000      { Az ES a képre mutat }
    mov  ax,Lsize        { Címszámítás a koordinátákból }
    mul  Y
    add  ax,X
    mov  Cadr,ax         { A kiszámított címet a Cadr-ba teszi }
    cld
    mov  dx,DS           { A DX az eredeti adatszegmensenet tárolja }
    lds  si,S            { A DS:SI a sztringre mutat }
    mov  bx,DS           { A BX a sztring szegmensét tárolja }
    lodsb               { A sztring első adata a hossza }
    mov  cl,al
    xor  ch,ch           { A hossz a CX-be kerül }
    jcxz @exit          { Kiugrás, ha a hossz nulla }
@cikl:
    push cx             { Tárolja a számlálót }
    mov  DS,bx
    lodsb              { Veszi a következő karaktert }
    push si             { Tárolja a forráscímet }
    mov  DS,dx          { A DS megkapja az eredeti értékét }
    call Char8x8        { Kiírja a karaktert }
    pop  si
    pop  cx             { A tárolt adatok visszahívása }
    loop @cikl         { Minden karaktert kiír a képre }
@exit:
    mov  DS,dx          { A DS visszakapja eredeti értékét }
End;
```

A fenti eljárásokhoz hasonlóan más méretű karaktereket is használhatunk, illetve azokhoz újabb megjelenítő rutinokat írhatunk. Érdemes megfigyelni a *Vspace* változó hatását. A *char8x8* a *Vspace* értékét egyszerűen hozzáadja a *Cadr* címmutatóhoz, ami azt jelenti, hogy bármilyen irányú kiírást meg tudunk valósítani, csupán a *Vspace* értékét kell megfelelően megválasztani. Függőleges kiírásnál az értéke legyen például 2560, azaz $8 \cdot 320$.

8.6. Képterületek tárolása és másolása

A képterületek tárolásával kapcsolatos eljárások a **graph** egységben az alábbiak: a *GetImage*, a *PutImage* és az *ImageSize* függvény. A következő három rutin bemutatja a tárolást és a másolást 256 szín esetében.

Az első eljárás, a *GetImage* az *X1,Y1* (bal felső) és az *X2,Y2* (jobb alsó) koordinátákkal megadott területet eltárolja a *Dest* típus nélküli változóban, amely tipikusan egy heapben lefoglalt területre mutató pointer szokott lenni. Működése a következő: a megadott koordináták alapján kiszámítjuk a grafikus ablak bal felső sarkának címét és a hosszúságokat. Ezt követően soronként kimásoljuk a képet a *Dest* paraméterrel megadott területre. Természetesen a másolás előtt gondoskodni kell a szükséges hossz megállapításáról és a memória lefoglalásáról, de erről majd később lesz szó.

```
Procedure GetImage (X1,Y1,X2,Y2:word; var Dest); Assembler;
```

Asm

```

push    DS          { Tárolja a DS-t }
cld

{ Elsőként kiszámítjuk a képcímet és a hosszúságokat }

mov     DS,SegA000   { A DS mutat a képre, ez lesz a másolás
                      forráscíme }

mov     ax,Lsize      { Címszámítás a már ismert módon }
mul     Y1
add     ax,X1
mov     si,ax         { Képcím az SI-be, DS:SI a képre mutat }
les     di,Dest       { ES:DI a célcímre mutat }
mov     bx,X2         { A vízszintes méret megállapítása }
sub     bx,X1
inc     bx            { Szélesség a BX-be }
mov     dx,Y2         { A függőleges méret megállapítása }
sub     dx,Y1
inc     dx            { Magasság a DX-be }
mov     ax,bx
stosw                { Legelső adatként a szélességet tároljuk
                      el }

mov     ax,dx
stosw                { A következő eltárolt adat a magasság }

{ A most következő @save_loop végzi a tulajdonképpeni másolást }

ve_loop:
mov     cx,bx         { CX-be teszi a vízszintes hosszúságot }
push    si
rep     movsb         { Egy pixelsor kimentése, másolás
                      [DS:SI]->[ES:DI] }

```



```

pop     si
add     si,Lsize      { Az SI (a forrás) a következő sorra mutat }
dec     dx            { A függőleges hosszúság csökkentése      }
jnz     @save_loop    { Minden sort kimásol                      }

pop     DS            { Visszatölti az eredeti DS-t, visszatérés }
End;

```

Látható, hogy a *GetImage* hasonló a *BAR* eljáráshoz, de itt nem memóriafeltöltést (*stosb*), hanem memóriamásolást (*movsb*) végeztünk.

Az elmentett területet a *PutImage* másolja vissza. Az eljárás valamivel összetettebb, mint a *GetImage*, mert itt meg tudunk adni valamilyen logikai kapcsolatot, amelyet az ablak minden egyes pixele és a kép pixelei között el kell végezni, és ennek az eredményét kell megjeleníteni. A lehetséges kapcsolatok a következők:

CopyPut	= 0;	normál másolás
NormalPut	= 0;	normál másolás
XorPut	= 1;	XOR kapcsolat a kép és az Image között
OrPut	= 2;	OR kapcsolat a kép és az Image között
AndPut	= 3;	AND kapcsolat a kép és az Image között
NotPut	= 4;	az Image pixeleit invertálja

A fenti öt művelet közül a normál másolás a leggyorsabb, és általában ezt használjuk a leggyakrabban. Ezért a *PutImage* külön kezeli ezt az esetet. A működése a címszámításig és a hosszúságok meghatározásáig megegyezik a *GetImage*-ével, utána pedig a logikai kapcsolatok miatt két ciklus lefutására nyílik lehetőség:

1. normál másolás esetén a *GetImage*-hez hasonló másolás, fordított adatiránnyal,
2. logikai kapcsolat esetén a pixelek a kirajzolás előtt logikai kapcsolatba kerülnek a képpel, és ennek eredményét jelenítjük meg.

Procedure PutImage(X,Y:word; **var** Source; Bop:word); **Assembler;**

Asm

```

push    DS            { Tárolja a DS-t                      }
cld

```

```

{ Az első lépés a képcím kiszámítása }

```

```

mov     ES,SegA000
mov     ax,Lsize
mul     Y
add     ax,X
mov     di,ax          { ES:DI a képernyőre mutat          }
lds     si,Source      { DS:SI pedig a forrásadatokra      }

```

```

    { A szélességet és a magasságot a GetImage már kiszámolta
      és eltárolta }

lodsw
mov    bx,ax          { Az első adat a szélesség, ezt a BX-be
                      tölti }

lodsw
mov    dx,ax          { A második adat a magasság, ez a DX-be
                      kerül }

    { Most megvizsgáljuk a logikai kapcsolat típusát }

mov    ax,Bop         { AX-be a kapcsolat típusa }
test   ax,ax          { Ha AX<>0, akkor valamilyen logikai kapcsolat
van    }
mov    ah,al
jnz    @logical_move  { Ugrás, ha van valamilyen logikai
                      kapcsolat }

    { A CopyPut, ill. a NormalPut egyszerű MOVE, ezért ez a
      leggyorsabb }

@normal_move:
mov    cx,bx          { Vízszintes hosszúság }
push   di
rep    movsb          { Kimásolja a képre a pixeleket }
pop    di
add    di,Lsize        { A célcím növelése (képcím) }
dec    dx              { DX darab vízszintes sort tárolunk }
jnz    @normal_move    { Minden sort kimásol a képre }
jmp    @exit           { Kilépés }

    { A logikai kapcsolatok miatt ez a rész lassúbb, mint az előző }

@logical_move:
push   di
mov    cx,bx          { A CX-be tölti a vízszintes hosszúságot }
@log1:
lodsb          { Egy adat a memóriából, a [DS:SI] címről }

    { Ugrás a logikai kapcsolatnak megfelelően }

cmp    ah,1
jz     @xorput         { AH=1 esetén KIZÁRÓ VAGY kapcsolat }
cmp    ah,2
jz     @orput          { AH=2 esetén VAGY kapcsolat }
cmp    ah,3
jz     @andput         { AH=3 esetén ÉS kapcsolat }

```



```

@notput:          { AH>=4 esetén NEM kapcsolat          }
    not    al
    jmp    @putpx
@xorput:
    xor    al,ES:[ di]    { XOR  }
    jmp    @putpx
@orput:
    or     al,ES:[ di]    { OR   }
    jmp    @putpx
@andput:
    and    al,ES:[ di]    { AND  }
@putpx:
    stosb          { Az adat kivitele a képre          }
    loop    @log1    { A @log1 ciklus kirak egy sort    }
    pop     di
    add     di,Lsize  { A képcím nő, rááll a következő sorra }
    dec     dx
    jnz     @logical_move { Minden sort kivisz          }
@exit:
    pop     DS        { Visszatölti az eredeti DS-t      }
End;

```

Az *ImageSize* függvény meghatározza az elmenteni kívánt terület memóriaigényét. A terület $(X2-X1+1)*(Y2-Y1+1)$ darab pixelből áll. Mivel a *GetImage* az első négy bájt a szélességet és a magasságot tárolja, ezért a hosszúság a fenti szorzat eredménye plusz négy. Az *ImageSize* függvénynek — ugyanúgy, mint a *GetImage*-nek és a *PutImage*-nek — fontos a koordináták sorrendje, azaz az *X1,Y1* mindig a bal felső, míg az *X2,Y2* mindig a jobb alsó sarokpontot jelöli.

Function ImageSize(*X1,Y1,X2,Y2:word*):word; **Assembler;**

Asm

```

    mov     ax,X2      { A vízszintes hosszúság kiszámítása          }
    sub     ax,X1
    inc     ax          { A szélesség az AX-be kerül                  }
    mov     dx,Y2      { A függőleges hosszúság kiszámítása          }
    sub     dx,Y1
    inc     dx          { A magasság a DX-be kerül                    }
    mul     dx          { Szélesség*magasság                          }
    add     ax,4        { +4 a szélesség és a magasság tárolásához    }

```

End;

Az *ImageSize* által megadott hosszúságú területet kell lefoglalni a heapben, mielőtt meghívánk a *GetImage* eljárást.

8.7. Ablakon belüli görgetés

A 256 színű módban az ablakon belüli görgetést is könnyebben el tudjuk végezni, mint a bitplane-ek esetében, de igazából sajnos ez a mód sem alkalmas az eltologatásokra, mert túl nagyok a másolandó memóriaterületek. Az egységben megtalálható a négyirányú pixeles eltolás, de látni fogjuk, hogy a rutinok csak kis ablakok esetén használhatók; nagy méreteknél lelassulnak. A görgetés működését már ismerjük. A megadott ablakon belül egy megadott irányba kell másolni a memóriát — amelynek bájtjai most pixeleket tartalmaznak. Mivel a négy rutin működése a címek kiszámításától eltekintve azonos, ezért itt most csak a felfelé eltoló *ScrollUp* eljárást nézzük meg.

Procedure ScrollUp(X1,Y1,X2,Y2:word); **Assembler;**

Asm

```

    push    DS
    mov     ES,SegA000
    mov     DS,SegA000    { A forrás- és a célcím is a képen van      }
    cld
    mov     ax,Lsize      { Címszámítás                          }
    mul     Y1
    mov     di,X1
    add     di,ax          { ES:DI most a célcímre mutat          }
    mov     si,di
    add     si,Lsize      { DS:SI most a forráscímre mutat          }
    mov     bx,X2
    sub     bx,X1
    inc     bx            { A vízszintes méret a BX-be kerül          }
    mov     dx,Y2
    sub     dx,Y1         { A függőleges méret pedig a DX-be          }
    mov     ax,Lsize
    sub     ax,bx         { A címnövelő érték az AX-be          }
@lp:
    mov     cx,bx         { Egy sor hossza a CX-be          }
    rep movsb             { Egy sor másolása          }

    add     si,ax
    add     di,ax         { A címek növelése          }
    dec     dx
    jnz     @lp           { Minden sort feljebb másol          }
    mov     cx,bx
    xor     al,al         { Az utolsó sort nullázza          }
    rep stosb
    pop     DS
End;
```


8.8. Egyéb eljárások

Az egységben még két kiegészítő eljárás található. A *Screen_Init* bekapcsolja a 320×200/256-os módot, a *ClearScreen* eljárással pedig törölni tudjuk a képet.

Procedure Screen_Init; **Assembler;**

Asm

```
mov ax,0013h      { 013h mód, 320x200/256 }
int 10h
```

End;

Procedure ClearScreen; **Assembler;**

Asm

```
cld
xor ax,ax          { Nulla adatokkal töltjük fel a képet      }
mov cx,32000       { 32 000 szó = 64 000 bájt                }
mov di,ax
mov ES,SegA000     { Az A000 szegmensben 64000 bájt          }
                   { kinullázása.                             }
rep stosw          { Word adatokkal gyorsabb, mint bájt      }
                   { adatokkal                                }

```

End;

END.

Ezzel a **GR256.PAS** egység ismertetésének a végére értünk. Nem állítjuk, hogy ez az egység helyettesíti a *graph* egységet, de a legfontosabb grafikus feladatokat már el tudjuk végezni, és az sem utolsó szempont, hogy gyors és kis helyigényű rutinokkal dolgozunk. Nem volt szó a teljes kép görgetéséről és az osztott képernyős üzemmódról, de ezekre a feladatokra a *normál bájt/pixel* formátumú kép a nagy címtartomány miatt nem alkalmas.

Jól látható, hogy a **grplus** egységben ismertetett rutinokhoz képest ezek a megoldások lényegesen egyszerűbbek. Ezért ha valamilyen grafikus programot kell készíteni, és a felbontásnak nem kell nagynak lennie, akkor érdekesebb a 256 színű grafikát alkalmazni, mint a bitplane technikájú 16 színűt.

A lemez mellékleten található **G256DEMO.PAS** program bemutatja az egység rutinjainak működését.



8.9. BOB-ok a 256 színű képen

Elérkeztünk a BOB-ok programozásához a 256 színű képen. Itt elhagyjuk a 16 színű módnál sorra vett lehetőségeket (hiszen igazán nem volna szép ezen a képen egyszínű BOB-okkal foglalkozni), és megpróbáljuk megoldani a legösszetettebb esetet, a többszínű és átfedő BOB-ok programozását. Sajnos a kép mérete miatt ebben az esetben nem lehet a kettősbuffer-technikát alkalmazni, hiszen egyetlen kép is lefoglalja a teljes 64K videomemóriát, a 128K videomemória megjelenítését pedig nem minden VGA kártya tudja elvégezni. Ezért itt másféle megoldást kell találnunk. Az igaz ugyan, hogy a videomemóriában csak egyetlen képet tudunk létrehozni, de ezt a képet a bitplane-ekkel szemben teljes egészében el tudjuk érni egyszerű memóriaolvasással és -írással. Azaz, ha a bufferképeket a videoterületen nem is tudjuk létrehozni, a normál memóriában ettől még lehetnek képek, csak ezeket mindig be kell másolni a képmemóriába. Csakhogy, ha a két bufferkép a normál memóriában van, akkor minden megjelenítéskor 64 000 bájtot kellene bemásolni az A000:0000 címre, és ez biztos, hogy lassú lenne. Ezért most az úgynevezett *folyamatos képfrissítés* technikáját fogjuk alkalmazni.

Hogyan működik ez a képfrissítés? Az elv hasonlít a kettősbuffer-technikához, de itt a kirajzolt kép megjelenítéséhez nem elegendő csupán a képcímet átállítani — hiszen ezt nem is tudjuk megtenni —, hanem a *változásokat* nekünk kell bemásolni a valódi képmemóriába. A kulcsszó a *változások* másolása. Tehát amikor frissítjük a képet, akkor nem az egész képet másoljuk be a látható területre, hanem csak azokat a részeit, amelyek megváltoztak. Hogyan zajlik le a másolás, és honnan állapítjuk meg, hogy hol vannak a változások? Ugyanúgy, mint a kettősbuffer-technikánál, itt is több képet tárolunk a memóriában. Először is kell egy eredeti kép, amelyet nem módosítunk, ez legyen az *OrgSCR*. Funkciója ugyanaz, mint a kettős buffernél a „C” kép funkciója, azaz tárolja az eredeti képernyőt, hogy a BOB-ok alatti területet innen vissza lehessen másolni, amikor a bufferképről el kell tüntetni a BOB-okat. Kell egy *Buf1SCR* munkakép, amelyen a tulajdonképpeni rajzolást végezzük. Végül kell egy *Buf2SCR* bufferkép, amely mindig az utolsó rajzolás állapotát tartalmazza. A frissítés folyamata a következő (egyelőre a BOB-ok nélkül):

A rajzot kirajzoljuk a *Buf1SCR* képre. Ezt követően a *Buf1SCR* képet pixelenként elkezdjük összehasonlítani a *Buf2SCR* képpel. Ahol eltérés van, ott új rajz található, ezért az eltérést ki kell rajzolni a valódi képre, és át kell másolni a *Buf1SCR*-ről a *Buf2SCR*-re. Így amikor az összehasonlítás véget ér, a két kép ismét ugyanazt az adatot fogja tartalmazni, és még a valódi kép is módosult a *Buf1SCR*-en végrehajtott változtatások miatt, tehát megjelent a módosítás. Innen kezdve ismételni kell a folyamatot. Újra lehet rajzolni a *Buf1SCR*-re, és újra lehet frissíteni.

A BOB-ok kirajzolásával együtt a folyamat a következő tevékenységekkel bővül:

1. A *BufI SCR*-ről le kell törölni a BOB-okat. Ez az *Org SCR*-ről való másolást jelenti.
2. El kell mozdítani a BOB-okat. Ezt a *MoveBOBs* rutin végzi.
3. Az új helyekre ki kell tenni a BOB-okat a *BufI SCR*-en.
4. A fent leírt frissítési folyamattal fel kell frissíteni a képet, ezáltal a BOB-ok a látható képen elmozdulnak.

Ha ezeket a részfeladatokat ezután ciklikusan ismételjük, akkor a BOB-ok a képen mozogni fognak. A módszer előnye, hogy nem elektronsugár függő, tehát nem szükséges várakozni. Másrészt viszont a módszer korlátozza a BOB-ok számát. A képen történő változások sok BOB esetén ugyanis egyre szaporodnak, és a frissítő résznek egyre több adatot kell mozgatnia, míg ez egy határ után túlságosan lelassítja a mozgást.

A következő példaprogram bemutatja a módszert működés közben. A képernyő és a BOB-ok is 256 színűek lehetnek. A BOB-ok színeinek korlátozásával nem tudunk gyorsítani, hiszen a BOB-ot mindenképpen bájtontként (pixelenként) kell kirajzolni, akár egyszínű, akár száz. Egy 16×16-os BOB esetén ez 256 bájt mozgatását jelenti. A tapasztalat szerint egy 12—16 MHz-es 286-os gép 16 darab 16×16-os BOB-ot még jól mozgat, de ennél többel már nem birkózik meg. A 33 MHz-es 486-os gép 32 BOB-nál még gyors, de több BOB esetén már ez is kezd lelassulni.



BOB256.PAS

```
(*****
(*)
(*) BOB-mozgatás a normál bájt/pixel formátumú 256 színű képen *)
(*)
(*****)
```

Uses crt;

Type

```
{ A memória és a memóriamutató típus definiálása }
MemPtr = ^MemArray;
MemArray = Array[ 0..65534] of byte;

{ Egy BOB adatmező }
BOB = Record
    X : word; { X koordináta ( [bx] ) }
    Y : word; { Y koordináta ( [bx+2] ) }
    M : word; { Mintaszám ( [bx+4] ) }
    A : word; { Képcím ( [bx+6] ) }
End;

{ Az M=0..255 lehet csak }
```

Const

```
BOBNum  = 31;           { Ennyi BOB lesz a képen }
Speed   = 1;           { Elmozdulás/sebesség   }
SegA000 : word = $A000; { A kép szegmenscíme   }
```

Var

```
i,j,k,l : integer;
f        : file;           { Általánosan használt változók }

pal      : array[ 0..767] of byte;   { A paletta tárolása }

BOBs     : array[ 0..BOBNum] of BOB; { BOB-adatok           }

dx       : array[ 0..BOBNum] of integer; { BOB-elmozdulás adatok }
dy       : array[ 0..BOBNum] of integer;

OrgScr   : MemPtr;         { Az eredeti kép           }
Buf1Scr  : MemPtr;         { Az első bufferkép    }
Buf2Scr  : MemPtr;         { A második bufferkép   }
Shapes  : MemPtr;         { Eredeti BOB-minták   }
```

A *LoadPic* eljárás egy kimentett VGA képet és palettát tölt be a videomemóriába, illetve a palettát feltölti a megadott színekkel. A lemezen található **vgascrn.00** fájl tartalmaz egy 64 000 bájtos képet (amelyet a **palanim.pas** program generált) és egy 768 bájtos palettát.

```
{ A háttérkép betöltése lemezről }
```

Procedure LoadPic;**Begin**

```
assign(f, 'vgascrn.00');           { A kép neve           }
reset(f, 1);
blockread(f, mem[ $a000:0], 64000); { Beolvasás a videomemóriába }
blockread(f, pal, 768);            { A paletta beolvasása   }
close(f);
```

asm

```
push es
mov  ax, 1012h           { A paletta beállítása a BIOS
                        rutinnal           }

xor  bx, bx
mov  cx, 256             { 256 színt állítunk be           }
push ds
pop  es
lea  dx, pal             { ES:DX adja a palettaadatok címét }
int  10h                 { A BIOS-beállító meghívása   }
pop  es
```

end;**End;**

@Scanning:

```

xor    al,al                { A Z bitet 1-be állítja }
repz   cmpsw                { Összehasonlítás, amíg egyezik a két
                             buffer }

jz     @exit                { Kiugrás, ha a végére ért }
mov     ax,DS:[ si-2]        { Az új adat az AX-be kerül }
mov     ES:[ di-2] ,ax       { Átmásolja a kettes bufferbe }
mov     dx,ES                { Az ES tárolása, amíg frissítjük a
                             képet }

mov     ES,bx                { Az ES most a képernyőre mutat }
mov     ES:[ di-2] ,ax       { Átmásolja a képre a változást }
mov     ES,dx                { Visszatölti az ES eredeti értékét }
jmp     @Scanning            { Folytatja az összehasonlítást }

```

@Exit:

```

pop     ds
End;
```

A *ClearBOBs* eljárás eltünteti a *BuflSCR*-ről a BOB-okat. A BOB-ok megjelenítésekor az utolsó képcímet az A BOB-mezőben tároljuk. Innen kell kivenni a címet, és az *OrgSCR*-ről a *BuflSCR*-re másolni a BOB alatti területet.

```
{ A BOB-ok eltüntetése a bufferképről, a háttér helyreállítása }
```

Procedure ClearBOBs; **Assembler;**

Asm

```

push    bp                { A BP-re is szükség lesz }
cld
mov     dx,DS              { A DX tárolja az aktuális adatszegmenst}
lds     si,OrgScr          { DS:SI mutat az OrgSCR-re }
mov     ax,DS              { Az AX tárolja az OrgSCR szegmensét }
mov     DS,dx
lea     bx,BOBs+6          { DS:BX a címeket adja }
les     di,BuflScr         { ES:DI a BuflSCR képre mutat }
xor     si,si
mov     cx,BOBNum+1        { Ennyi BOB van }

```

```
{ A @ClrCikl letörli a BOB-okat }
```

@ClrCikl:

```

push    cx
mov     DS,dx
mov     di,[ bx]           { A BOB adatrekordjából kiveszi az
                             A mezőt }

mov     si,di              { SI és DI megkapja a megfelelő címet }
add     bx,8               { BX növelése, ráállítás a következő
                             BOB adatra }

mov     DS,ax              { DS most már az OrgSCR-re mutat }
mov     bp,16              { Egy BOB 16 pixel magas }

```



```
{ A @BOBlines állít helyre egy BOB alatti területet }
```

```
@BOBLines:
```

```
mov    cx,8
rep movsw           { 8 db word, azaz 16 pixel átmásolása }
add     di,320-16
add     si,320-16    { Mindkét címnek nőnie kell }
dec     bp
jnz     @BOBLines   { Minden sort töröl }
pop     cx
loop    @ClrCikl     { Minden BOB-helyet töröl }
mov     DS,dx        { Visszaállítja a DS-t }
pop     bp           { Visszatölti a BP-t }
```

```
End;
```

A *PutBOBs* eljárás a BOB-okat kirakja a *BuflSCR* képre. A BOB-ok kirajzolását a már ismert módszerrel hajtjuk végre. Az *X,Y* koordináták alapján kiszámoljuk a képcímet, az *M* mintakód alapján pedig a megfelelő *Shape* címet. A címek kiszámítása után a BOB-ot pixelesen ki kell másolni a képre. A Shape adatokban levő nulla esetén a háttér jelenik meg, azaz a nulla szín nem lehet a BOB-ban.

```
{ A BOB-ok kirakása a bufferképre }
```

```
Procedure PutBOBs; Assembler;
```

```
Asm
```

```
push    bp
push    ds
cld

les     di,BuflScr    { Az ES:DI a BuflSCR képre mutat }

mov     ax,DS
lds     si,Shapes     { DS:SI fogja szolgáltatni a forrásadatokat
                      (Shape) }
mov     bp,DS         { A BP tárolja a Shape szegmenscímet }
mov     DS,ax
lea     bx,BOBs       { DS:BX adja az adatokat }
```

```
{ Ez a ciklus végzi a BOB-ok kirajzolását }
```

```
mov     cx,BOBNum+1
```

```
@PutCikl:
```

```
mov     ax,seg @data
mov     DS,ax         { Ezek az adatok az adatszegmensben
                      vannak }
mov     ax,320        { A képcímszámítás következik }
mul     word ptr [bx+2] { 320*Y }
add     ax,[bx]       { 320*Y + X ez a képcím }
```

```

mov    [ bx+6] ,ax      { A címet tárolja az A mezőben      }
mov    di,ax            { Az ES:DI most a BuflSCR képre      }
                        mutat
mov    ax,[ bx+4]       { AX-be a mintaszám, amely 0..255      }
                        lehet csak
add    bx,8            { A BX a következő BOB-adatra mutat}
xchg   al,ah           { Ez a csere 256-tal való szorzást      }
                        eredményez
mov    si,ax           { SI a mintaoffsetet adja          }
mov    DS,bp           { Most ES:DI az egyes bufferre,       }
                        { DS:SI pedig a megfelelő mintára     }
                        mutat

      { Most pixelenként kirakjuk a BOB-ot }
mov    dl,16           { DL számolja a sorokat              }
@Lines:
mov    dh,16           { DH számolja a sorokon belüli       }
                        pixeleket
@Pixels:
lodsb           { Egy bájt lehívása a Shape adatokból }
and    al,al
jnz    @Put       { Ugrás, ha nem háttér színű              }
mov    al,ES:[ di]   { Háttérszín esetén önmaga kerül vissza }
@Put:
stosb           { A pixel kirakása a képre                  }
dec    dh        { A pixelszámláló csökkentése              }
jnz    @Pixels    { Visszaugrás, ha a sorban van még        }
                        pixel
add    di,320-16     { Címnövelés                            }
dec    dl          { A sorszámláló csökkentése              }
jnz    @Lines       { Visszaugrik, ha van még sor            }

loop   @PutCikl      { Minden BOB-ot kirak a képre          }

pop    ds
pop    bp
End;

```

A *MoveBOBs* végzi a BOB-ok mozgatását az egyszerű, falról lepattanó mozgás megvalósításával.

Procedure MoveBOBs;

Begin

```

for i:=0 to BOBnum do { Az összes BOB elmozdítása }
  with BOBs[ i] do

```


Begin

```
inc(x,dx[i]*Speed); { A koordináták növelése }
inc(y,dy[i]*Speed);
{ A növekmény invertálása, ha elértük a falat }
if (x<3) or (x>302) then dx[i] := -dx[i];
if (y<3) or (y>182) then dy[i] := -dy[i];
```

End;

End;

A főprogramban inicializáljuk a grafikát, lefoglaljuk a memóriaterületeket és az adatokat betöltjük a lemezről.

BEGIN

```
GraphOn; { A grafika bekapcsolása, helyfoglalások }
```

```
Getmem(Shapes,2048); { A BOB-minták helye, 16 BOB-Shape }
```

```
{ A Shape-ek adatainak beolvasása }
```

```
assign(f,'bob256.dat');
```

```
reset(f,1);
```

```
blockread(f,Shapes^,2048);
```

```
close(f);
```

```
{ Képgenerálás }
```

```
LoadPic; { Az eltárolt kép beolvasása }
```

```
move( mem[ $a000:0],OrgScr^,64000); { Az eredeti kép feltöltése }
```

```
move( mem[ $a000:0],Buf1Scr^,64000); { Az egyes buffer feltöltése }
```

```
move( mem[ $a000:0],Buf2Scr^,64000); { A kettes buffer feltöltése }
```

```
{ A BOB-ok adatainak véletlenszerű feltöltése }
```

```
for i:=0 to BOBnum do
```

```
  begin
```

```
    dx[i] := (random(2)*2) -1 ;
```

```
    dy[i] := (random(2)*2) -1 ;
```

```
    with BOBs[i] do
```

```
      Begin
```

```
        x:=10+i*8;y:=10+random(150);
```

```
        m:=i and 7;
```

```
      End;
```

```
  end;
```

```
{ Az alapállapot megjelenítése (még nincs mozgás) }
```

```
PutBOBs;
```

```
ScreenFresh;
```

```
readln; { Várakozás az ENTER-re }
```

```

{ Ez a ciklus végzi a mozgatót és a megjelenítést }
repeat
  ClearBOBs;      { A régi helyek regenerálása }
  MoveBOBs;       { A BOB-ok elmozdítása }
  PutBOBs;        { A BOB-ok kirakása az új helyekre }
  ScreenFresh;    { Képfrissítés, megjelenítés }
until keypressed;
while keypressed do readkey;
{ Egy billentyűnyomásra kilép }

Freemem(Shapes,4096); { A Shape adatterület felszabadítása }

GraphOff; { Visszatérés karakteres módba, a memória felszabadítása }
END.

```

A *normál bájt/pixel* felépítésű képpel való munkát ezzel a BOB-programmal be is fejeztük. Tudunk a képre pixeleket, vonalakat, téglalapokat rajzolni, tudunk területeket menteni és ablakot görgetni. Megnéztük a BOB-ok kirajzolását a *képfrissítéses* módszerrel, és láthattuk a képszerkezet korlátait is. Az egyes feladatokat még tovább lehetne elemezgetni és optimalizálni, de ez már legyen az Olvasó feladata! Összességében, aki megértette a fejezet példaprogramjait és a működési elveket, az tudja, hogy mire érdemes és mire nem érdemes használni ezt a VGA módot.

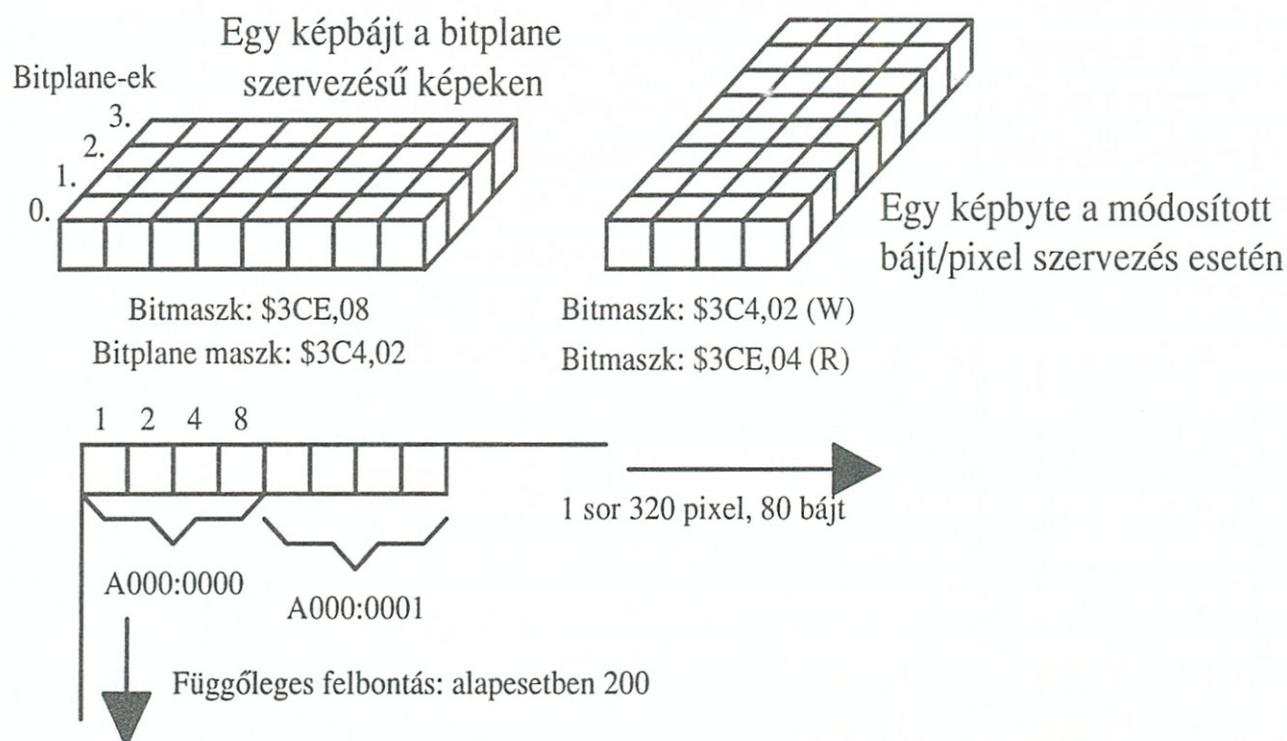
9. A 256 színű grafikus mód programozása II. rész



AG256MODE

Az alábbi fejezet egy olyan videomódról szól, amelynek a lehetősége megvan ugyan, de a VGA-BIOS nem támogatja. Úgy is lehetne mondani, hogy egy nem szabványos videomódról lesz szó, amelyet az angol irodalom *register-compatibility only video mode* néven emleget, azaz csak regiszter-kompatibilis, de a működése nem garantált. Mindezek ellenére foglalkozunk vele, mert a VGA kártyák túlnyomó része ismeri és használni is tudja. Mivel azonban ez egy nagy memóriaigényű mód, ezért nagyobb az esély a működésére egy olyan kártyán, amelyen legalább 512K memória van, míg a 256K normál VGA esetében csak korlátozottan, vagy egyáltalán nem lehet használni. Ehhez a videomódhoz egy különleges, az előzőekben már említett, *módosított bájt/pixel* képernyőszerkezet tartozik, ismerkedjünk meg hát ezzel!

9.1. A módosított bájt/pixel képszerkezet



45. ábra A módosított bájt/pixel képszerkezet

Ez a képszerkezet kombinálja a *bitplane* és a *bájt/pixel* formátumot, és ily módon különleges képfelépítés jön létre. Nevezetesen: a képernyőn egy memóriacímen 4 pixel található, és mindegyik pixel 256 színű lehet. Az egészet úgy kell elképzelni, mintha a színes bájtok merőlegesen elfordultak volna, és a keskenyebb oldalukat mutatnák felénk.

Tehát a CPU címezése szempontjából egyetlen bájton négy pixel található. A pixeleket a *\$3C4,02 Plane Write Enable* regiszterrel tudjuk kiválasztani írásra, míg olvasáskor a *\$3CE,\$04 Read Plane Select* regiszterrel kell megadni az olvasni kívánt pixel számát. Ráadásul ez a „négybites bájt” még fordítva is áll, tehát a nulladik bit van a bal szélén és a harmadik bit a jobb szélén. Az biztos, hogy egynéhány alapfeladattal nehéz lesz megbirkózni. Hogy miért használjuk mégis ezt a nyakatekert, agyonbonyolított képet, annak az az oka, hogy bizonyos feladatok 256 színben csak ezzel a képszerkezettel oldhatók meg. Nézzük meg az előnyeit:

1. A címtartomány a negyedére csökken. Mivel egy CPU címen 4 pixelt, azaz négy bájtot lehet elérni, ezért az általunk megcímezhető képmemória megnégyszereződik. A címtartomány csökkenését úgy érzékeljük, mintha egy képernyő (a normál 320×200-as felbontásnál) nem 64 000 (320*200), hanem csak 16 000 (80*200) bájt memóriát igényelne. A valóságban persze a kép memóriaigénye ugyanannyi marad, de ezt a memóriát már a VGA kártya kezeli, nekünk nem kell törődni vele.
2. Osztott képernyő lehetősége. A címtartomány megnövelésével az osztott képernyős módot is be tudjuk állítani. Ezt a *normál bájt/pixel* szerkezet esetén is meg lehetne tenni, de ott egyszerűen nincs már videomemória két különálló kép kezelésére.
3. A görgetés lehetősége. A teljes képernyő görgetésének lehetőségeit ki tudjuk használni a címtartomány csökkenése miatt, hiszen létrehozhatunk több képernyőlapot, amit a *normál bájt/pixel* formátumnál nem tudtunk megtenni.
4. A felbontás növelésének lehetősége. A címtartomány csökkenése miatt lehetőségünk van a felbontás növelésére. A videomemória ebben az esetben egy bájton négy pixelt tárol, tehát egy rastersorhoz — 320 pixeles vízszintes felbontás esetén — 80 bájt memória kell, ami azt jelenti, hogy a teljes 64K-ban 65 536/80, azaz 819 rastersort tudunk felépíteni. Bár a VGA kártya megengedi a vízszintes felbontás növelését is (a 320-at 360-ra lehet növelni), ez csak kis mértékben növeli a kép méretét, ezért inkább a függőleges felbontás növelésével foglalkozunk.
5. BOB-oknál a kettős buffer lehetősége. A több képernyőlap miatt alkalmazni tudjuk a kettősbuffer-technikát, ami a BOB-ok programozása szempontjából e képszerkezet igen előnyös tulajdonsága.
6. Színes bájtok összemontírozásának lehetősége. A *módosított bájt/pixel* formátumnak ez a legelőnyösebb tulajdonsága, hiszen az összemontírozást eddig még egyetlen képszerkezetnél sem tehettük meg. Itt viszont úgy lehet összemásolni két színes bájtot, hogy egy megadott maszk szerint vezérelni tudjuk, hol jelenjenek meg a két bájt adott pixeli. Ez lehetőséget ad számunkra, hogy BOB-okat tároljunk a képmemóriában, és ennek további két előnyös tulajdonság lesz a következménye. Egyrészt a BOB-ok nem foglal-

nak le helyet a normál memóriában, másrészt csökken a BOB-ok látszólagos hossza. Ha egy BOB 16×16 pixeles és 256 színű, akkor 256 bájt szükséges az eltárolásához, ha viszont ezt a képmemóriában tároljuk, akkor a CPU szempontjából ez csak 64 bájt területnek látszik.

9.2. A képszerkezethez tartozó unit

A *normál bájt/pixel* formátumhoz hasonlóan itt is egy egységbe foglalva tárgyaljuk a legfontosabb feladatok megoldását. Az egység a lemezmellékleten **GR256_2.PAS** néven szerepel. Mivel ezt a képformátumot sehol sem ismertetik, és nincs is hozzá semmilyen segédprogram (pl. .BGI interfész), ezért itt valóban minden egyes grafikus alapeljárást magunknak kell megírnunk.



GR256_2.PAS

```
{ $G+}
{ $L char8x8.obj}
```

```
UNIT GR256_2;
```

Az egység *interface* részében deklaráljuk azokat a változókat és konstansokat, amelyeket később majd kívülről is el lehet érni. A változók feladatait az azokhoz tartozó megfelelő eljárásoknál nézzük majd meg részletesen.

INTERFACE

Const

```
Lsize = 80;                { A kép logikai hossza          }
SegA000 : word = $a000;    { A képernyő szegmense      }

{ Színdefiníciók a keretezett téglalapokhoz }
Cols      : Array[ 0..11] of byte=(15,8,7,8,15,7,15,8,3,8,15,3);

{ Pixelpozíciók egy képbájton belül a PutPixel számára }
PixelPos  : Array [ 0..3] of byte = ($11,$22,$44,$88);

Ink : byte = 15;    { A szövegírás és a rajzolás színe }
Paper : byte = 01;  { Papírszín a szövegíráshoz       }
Over : byte = 01;   { Papír/háttér kiválasztása        }

Yres : word = 200;  { A függőleges felbontás           }

Cadr : word = 00;   { Képcím a karakterkiíráshoz       }
Cbit : byte = $11;  { Bitmaszk a karakterkiíráshoz     }
```


Var

```

{ A keretezett téglalapok színei }
Light : byte; { A bal és a felső oldal }
Dark  : byte; { A jobb és az alsó oldal }
Back  : byte; { A téglalap alapszíne }

```

Az eljárások több kisebb csoportra bonthatók a témakörök szerint:

1. Grafikus módba kapcsoló eljárások. Mint azt később majd látni fogjuk, ebben a képformátumban az összes szabványos függőleges felbontást be lehet állítani. A beállítást a CRTC regiszterek közvetlen programozásával érhetjük el, erről részletesebben a *Nem szabványos videomódok* fejezetben lesz majd szó. Itt elég csupán annyit tudni, hogy ezek a felbontások beállíthatók, és az aktuális függőleges felbontást az *Yres* változó tartalmazza.

```

Procedure _320x200;
Procedure _320x240;
Procedure _320x350;
Procedure _320x400;
Procedure _320x480;

```

2. Pixelrajzolás és lekérdezés.

```

Procedure PutPixel(X,Y:word;Color:byte); { Egy pixel kigyújtása }
Function GetPixel(X,Y:word):byte; { Egy pixel lekérdezése }

```

3. Vonalrajzolás. Az általános vonalrajzoló eljáráson kívül megoldottuk a függőleges és a vízszintes vonalak rajzolását is, ami természetesen gyorsabb, mint a normál *Line*.

```

Procedure Line(x1,y1,x2,y2:word); { Általános vonal rajzolása }
Procedure Vline(X,Y,Long:word); { Függőleges vonal rajzolása }
Procedure Hline(X,Y,Long:word); { Vízszintes vonal rajzolása }

```

4. Téglalaprajzolás. Négyféle téglalapot tudunk rajzolni: üreset (csak egy keretvonal), kitöltöttet, vékonyan keretezettet és vastagon keretezettet.

```

Procedure Box(X1,Y1,X2,Y2:word); { Üres téglalap rajzolása }
Procedure Bar(X1,Y1,X2,Y2:word); { Kitöltött téglalap rajzo-
                                     lása }
Procedure Bbar(X1,Y1,X2,Y2,Ty:word); { Vastagon keretezett tégla-
                                     lap rajzolása }
Procedure Cbar(X1,Y1,X2,Y2,Ty:word); { Vékonyan keretezett tégla-
                                     lap rajzolása }

```


5. Képterület tárolása és másolása. Az itt megvalósított eljárások a képernyő szerkezete miatt másképp működnek, mint a *graph* egység ugyanezen eljárásai. Nevezetesen ebben az esetben az X irányban nem pixelesen, hanem képbájtontként lehet megadni a méretet. Ez azért van így, mert ezen a képszerkezeten bizony elég nehéz a *PutImage* és a *GetImage* megvalósítása, és mi az egyszerűbb esetet választottuk.

```

Procedure GetImage(X1,Y1,X2,Y2:word; var Dest); { Egy ablak eltárolása }
Procedure PutImage(X,Y:word; var Source); { Egy tárolt ablak visszamásolása }
Function ImageSize(X1,Y1,X2,Y2:word):word; { Egy ablak méretének megállapítása }

```

6. Szövegkiírás. A két eljárás pixeles pozícióba tud írni 8×8-as karaktereket, és az előzőekben már megismert *Over* változó segítségével a háttér átírását tudjuk vezérelni.

```

Procedure OutCharXY(X,Y:word;C:char); { Egy karakter kiírása }
Procedure OutStrXY(X,Y:word; S:string); { Egy sztring kiírása }

```

7. Egyéb eljárások. A *SegRGBreg* egy RGB regiszter beállítását végzi, míg a *GetRGBreg* lekérdez egy RGB regisztert. A *ClearSegA000* törli a teljes képmemóriát.

```

Procedure SetRGBreg(regnum,Red,Green,Blue:byte); {Egy RGB regiszter beállítása }
Procedure GetRGBreg(Regnum:byte; var RGB ); {Egy RGB regiszter lekérdezése }
Procedure ClearSegA000; { A képmemória törlése }

```

Az összes rajzoló eljárás az *INK* változóval beállított színnel rajzol, kivéve a keretezett téglalapokat, ahol a típusdefiníció határozza meg a rajzolás színét.

Az *implementation* rész elején szerepel néhány olyan változó, amelyeket elrejtünk a külvilág elől.

IMPLEMENTATION

Const

```

{ A 350-es függőleges felbontás CRTC értékei }
_Data_350 : Array [ 0..7] of byte=($83,$85,$5D,$28,$0F,$63,$BA,$e3);
{ A 480-as és a 240-es függőleges felbontás CRTC értékei }
_Data_480 : Array[ 0..17] of byte=($0b,$3e,$00,$40,$00,$00,$00,$00,
                                     $00,$00,$ea,$8c,$df,$28,$00,$e7,
                                     $04,$e3);

```



```
{ Maszkbájtok a Hline és a Bar rutin számára }
  Fmask      : Array [ 0..3] of byte = ($0f,$0e,$0c,$08);
  Lmask      : Array [ 0..3] of byte = ($01,$03,$07,$0f);
```

Var

```
{ Regiszterek tárolására használt változók }
r0,r1,r2,r3 : word;
r4,r5,r6,r7 : word;

  LineColor : byte; { A vonalhúzás színe }
  LineBITpt : byte; { Vonalhúzás bitmaszk }
```

```
{ 8x8-as karakterkészlet a char8x8 rutin számára }
```

```
Procedure Charset; External;
```

Ezek után nézzük meg a feladatok megvalósítását a fenti témakörök szerinti felosztással!

9.3. A videomód beállítása

Az itt szereplő eljárások beállítják a *módosított bájt/pixel* formátumot és a kívánt függőleges felbontást. Mivel a felbontások módosításáról majd később lesz szó, és a rutinok csak a függőleges felbontás beállításában térnek el egymástól, ezért itt most csak egy eljárást nézünk meg részletesen.

```
Procedure _320x200; Assembler;
```

Asm

```
  mov  ax,0013h { Beállítjuk a normál 320x200/256-os módot }
  int  10h      { a BIOS módbeállító funkciójával }
  mov  dx,03D4h
  mov  ax,0014h { A 6. bitet töröljük a $3D4,$14 regiszterben }
  out  dx,ax
  mov  ax,0e317h { Beállítjuk a 6. bitet a word címzés }
  out  dx,ax     { kikapcsolására }
  mov  dx,03C4h
  mov  ax,0604h { Töröljük a 3. bitet }
  out  dx,ax
  call ClearSegA000 { A videobuffer törlése }
  mov  Yres,200 { Az Y felbontás jelzése az Yres változóban }
```

```
End;
```

Mint láthatjuk, a *módosított bájt/pixel* formátum beállításához három bit átállítására van szükség.

1. A *\$3D4, \$14 Underline Location* CRTC regiszter 6. bitje dupla szó (*doubleword*) címezsmódot határoz meg. Ez van beállítva a *normál bájt/pixel* formátum esetén, ekkor a regiszter értéke \$40 (01000000b). Elsőként tehát ezt a bitet kell törölni, ezért kerül a \$14-es regiszterbe 00h.
2. A *\$3D4, \$17 Mode Control* CRTC regiszter 6. bitje némely VGA kártyán a szó (*word*) címezsmódot jelzi, ha törölve van. Ez nem minden VGA kártya esetében van így, de pl. a **Trident 8900c** kártyán törölni kell ezt a bitet. A \$3D4, \$17 regiszter alapbeállításban \$A3-at (10100011b) tartalmaz, ezért mi (mivel bájt címezsmódot fogunk használni) \$E3-ba (11100011b) állítjuk.
3. A *\$3C4, \$04 Memory Mode* regiszter 3. bitjével tudjuk megadni azt, hogy a bitplane-eket a memóriacím 0—1. bitjei vagy pedig a bitplane-vezérlő regiszterek választják-e ki. Mivel itt a bitplane-vezérlő regiszterekkel fogjuk kiválasztani a pixeleket, ezért ezt a bitet törölni kell.

Ennek a három bitnek a beállítása a *módosított bájt/pixel* formátum bekapcsolását eredményezi, ezt követően töröljük a videomemóriát, és beállítjuk a függőleges felbontást jelző *Yres* változót. A többi módbeállító eljárás is ugyanezt teszi a CRTC regiszterek átprogramozásával kiegészítve, ami a függőleges felbontás beállítására szolgál.

```

Procedure _320x350; Assembler; {320x350/256 módosított bájt/pixel
                                képformátum                                }
Procedure _320x400; Assembler; {320x400/256 módosított bájt/pixel
                                képformátum                                }
Procedure _320x480; Assembler; {320x480/256 módosított bájt/pixel
                                képformátum                                }
Procedure _320x240; Assembler; {320x240/256 módosított bájt/pixel
                                képformátum                                }

```

9.4. Pont kirajzolása és lekérdezése

9.4.1. Egy pixel kigyújtása

A pixel kigyújtásához meg kell határozni az *X,Y* koordinátákhoz tartozó képcímet és a bitpozíciót. Mivel itt egy képbájt négy pixelt tartalmaz, ezért a bitpozíciót az *X* koordináta alsó két bitje választja ki a következőképpen:

X and 3	Bitszám	Bitmaszk
00b	00h	01h
01b	01h	02h
10b	02h	04h
11b	03h	08h

A fenti pozíció—bitmaszk összerendelést a *PixelPos* tömbbel valósítjuk meg. Ha megnézzük a tömbelemet, akkor látható, hogy a bájtok felső négy bitjében is kijelöltük ugyanazt a pozíciót, amelyet az alsó négy biten. Erre azért van szükség, mert a későbbi *Line* rutin is használja majd az itt lévő értékeket, és ott bitforgatás is lesz. Ha a két tetrád ugyanazt az értéket tartalmazza, akkor egy egyszerű **rol** vagy **ror** utasítással elérhetjük, hogy az alsó négy bit úgy forogjon, mintha a bit a harmadik bitből a nulladikba forogna ki és fordítva.

A képcím a már ismert $Lsize*Y+X$ képlettel számítható, de mivel az X alsó két bitje itt bitpozíciót határoz meg, az X -et el kell tolni két bittel jobbra. A képcím kiszámítása tehát a

$$képcím=Y*Lsize+(X \text{ shr } 2)$$

képlet alapján történik, ahol az $Lsize$ $320/4$, tehát 80. Ha a képcím és a bitpozíció is megvan, akkor a bitpozíciót ki kell vinni a $\$3C4, \$02$ Write Plane Enable regiszterbe, és a Color színt be kell írni a megfelelő képbájtba.

Procedure PutPixel(X,Y :word;Color:byte); **Assembler;**

Asm

```

mov     ES,SegA000      { Az ES-t ráállítjuk a képre           }
mov     ax,Lsize        { Címszámítás                         }
mul     Y               { AX = 80 * Y                         }
mov     di,X            { DI = X                             }
mov     bx,di           { Az X-et a BX-be is bevisszük        }
shr     di,2            { Az X eltolása két bittel           }
add     di,ax           { A DI most a képcímet tartalmazza    }
and     bx,3            { A BX AND 3 megadja a bitpozíciót    }
add     bx,offset PixelPos { A BX most a megfelelő PixelPos  }
                                elemre mutat                  }

mov     dx,03c4h
mov     al,02h          { A $03C4, $02 regisztert kell írni   }
mov     ah,[bx]         { AH-ba a bitmaszk a PixelPos táblázat- }
                                ból                             }

out     dx,ax           { A megfelelő bitplane engedélyezése  }
                                írásra                         }

mov     al,Color
mov     ES:[di],al      { A pixel kigyújtása a Color színnel  }

```

End;

9.4.2. Egy pixel színének lekérdezése

A képpont színének megállapításához a kigyújtáshoz hasonlóan szükségünk van egy képcímre és egy bitpozícióra. A rutin első része tehát itt is cím- és bitpozíciószámítás. A lekérdezéshez azonban nem a *\$3C4*, *\$02 Write Plane Enable* regisztert, hanem a *\$3CE*, *\$04 Read Plane Select* regisztert kell használni. Mivel ez a regiszter nem plane maszkot, hanem plane sorszámot tartalmaz, ezért itt nincs szükség a *PixelPos* táblázatra, hanem egyszerűen az *X AND 3* értéket kell kivinni a regiszterbe. A regiszter kiírása után a képbájt leolvasásakor a CPU megkapja a pixel színét.

Function GetPixel(X,Y:word):byte; **Assembler;**

Asm

```

mov     ES,SegA000      { Az ES a képre fog mutatni           }
mov     ax,Lsize        { Címszámítás                         }
mul     Y
mov     bx,ax           { Az Y*Lsize értéket a BX tárolja     }
mov     ax,X            { Az AX-be beolvassuk az X koordinátát }
mov     di,ax           { A DI-be is betesszük az X-et         }
shr     di,2            { Az alsó két bitre nincs szükség     }
add     di,bx           { DI most a képcímet tartalmazza      }
mov     dx,03ceh
and     al,3            { Az X alsó két bitje az AL-ben       }
mov     ah,al           { Az AL átmásolása az AH-ba          }
mov     al,4            { A 04-es vezérlőregiszter kiválasztása }
out     dx,ax           { Kivisszük az olvasni kívánt plane   }
                        számát
mov     al,ES:[di]      { Beolvassuk a pixel színét a képről  }

```

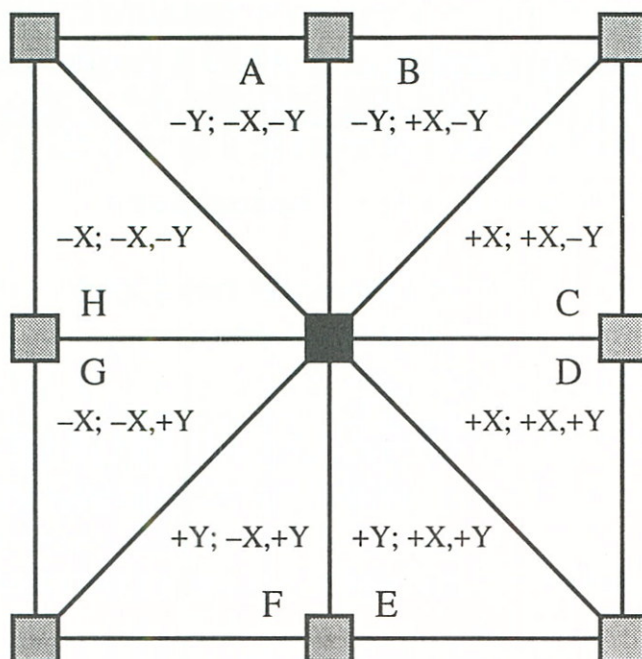
End;

9.5. Vonalrajzolás

9.5.1. Tetszőleges irányú vonal rajzolása

A vonalrajzolás elvéről többször is volt már szó, megismerhettük az elvet a 16 színű módnál ismertetett *Line Pascal* eljárásban, és láthattunk assembly nyelvű megvalósítást is a *normál bájt/pixel* képformátumnál. Az itt következő algoritmus eléggé összetett és hosszú lesz az előző vonalrajzoló eljárásokhoz képest, de a képszerkezet jellege miatt választani kellett, hogy rövid, de lassú, vagy terjedelmes, ám gyors vonalrajzolóval valósítsunk-e meg.

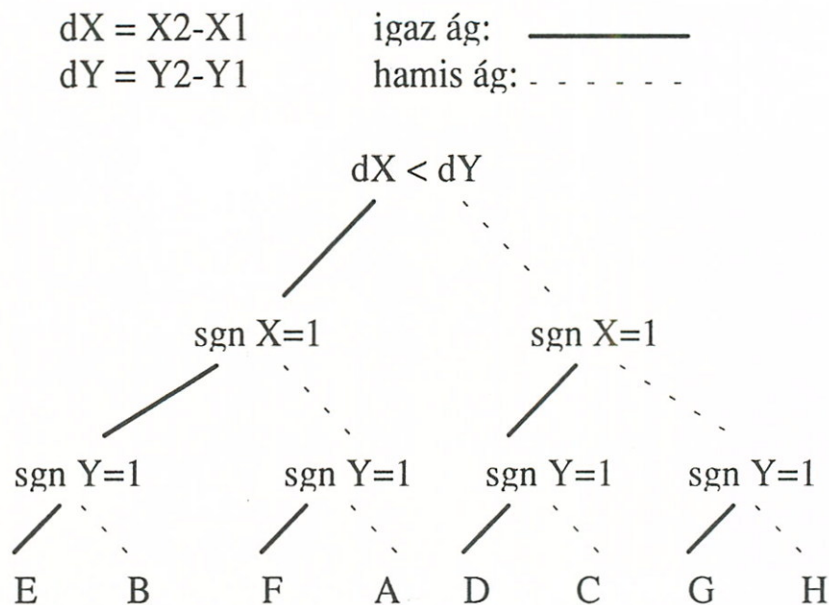
A vonalrajzolásnál ki kell választanunk a főirányt és az átlós irányt. Ez azt jelenti, hogy meg kell határozni azt az irányt — vízszintesen vagy függőlegesen —, amerre egyesével fogunk lépni, és meg kell állapítani az átló irányú elmozdulást, amerre csak minden n -edik főirányú lépés után lépünk. A két adatot az **abs X-abs Y** viszony, illetve az **sgn X** és **sgn Y** értékek adják meg. Mivel ez három feltétel, ebből nyolc különböző vonalhúzást kaphatunk meg. A 46. ábra a nyolc különböző esetet szemlélteti:



46. ábra A vonalhúzás nyolc lehetséges esete

Az ábra közepén levő fekete négyzet a kezdőpontot jelzi, míg a szélső világosabb négyzetek a végpont helyzetét mutatják a kezdőponthoz képest. Az A—H mezők a nyolc különböző rutin „működési mezőit” jelzik. A mezőkben található paraméterek pedig a fő- és az átlós irányt reprezentálják. Például a C mező $+X; +X, -Y$ paramétere azt jelenti, hogy a főirány pozitív X, tehát balról jobbra, míg az átlós irány $+X, -Y$, tehát jobbra és felfelé. Hogy mindez hogyan kapcsolódik az **abs X-abs Y** viszonyhoz és az **sgn X, sgn Y** értékhez, azt a 47. ábrán látható logikai gráf szemlélteti.

Tehát hogyan is fog lezajlani a vonalhúzás? A három feltétel alapján eldöntjük, hogy melyik ágot kell végrehajtani (A—H), és a megfelelő részrutin meghívásával kirajzoljuk a vonalat. Ez azt jelenti, hogy nyolc meglehetősen hasonló eljárást kell írni, amelyek csak az irányok miatt különböznek kissé egymástól. Mivel mindegyik csak egyetlen különleges vonalat rajzol, meglehetősen gyorsak. El kell még mondani azt is, hogy a függőleges és a vízszintes vonalakat itt nem választjuk külön a többitől, hanem ezekre majd külön eljárásokat írunk. (Természetesen ez a rutin is tud vízszintes és függőleges vonalakat húzni — hiszen az algoritmus univerzális —, de a későbbi *Vline* és *Hline* gyorsabb.)



47. ábra A vonalhúzás logikai szemléltetése

A részrutinok meghatározása a következő táblázat alapján történik:

Mező	$dX < dY$	$\text{sgn } X=1$	$\text{sgn } Y=1$
A	1	0	0
B	1	1	0
C	0	1	0
D	0	1	1
E	1	1	1
F	1	0	1
G	0	0	1
H	0	0	0

A *Line* eljárás tehát a fenti három feltétel alapján meghatároz egy hárombites értéket, amely szerint a megfelelő rutinra ugunk. Az ugrást egy táblázat alapján hajtjuk végre, amely a részrutinok címét tartalmazza.

```

Const LrutTab: Array[ 0..7] of word=(
  ofs(Line_H),ofs(Line_G),ofs(Line_C),ofs(Line_D),
  ofs(Line_A),ofs(Line_F),ofs(Line_B),ofs(Line_E) );

```

A *Line* eljárás elsőként kirajzolja a kezdőpontot. Ez a *PutPixel* meghívásával történik, amely a következőket végzi el:

- az ES-t ráállítja az A000 képszegmensre, amelyre később szükség lesz,
- a DX-be betölti a \$03C4 értéket, ami a bitmaszkoláshoz kell majd,
- a DI-be a kezdőpont képcímét teszi, ez is kell majd később.

Procedure Line(x1,y1,x2,y2:word); **Assembler;**

Asm

```

push  x1           { Kirajzoljuk a kezdőpontot           }
push  y1
mov    al,Ink
mov    LineColor,al { Tároljuk a rajzolás színét           }
push  ax
call   PutPixel     { Az első pont kirajzolása a PutPixel
                    rutinnal                               }
mov    LineBITpt,ah { A bitpozíció tárolása                               }
{ A képcím most a DI-ben van, az ES rámutat a képre, és a
  DX=03C4h                               }

```

Most meghatározzuk a hárombites értéket, amely majd a megfelelő rutint mutatja.

```

xor    bx,bx        { BX-ben lesz a logikai gráf eredménye,
                    az ugrás címe                               }
mov    ax,y1         { Az AX-ben létrehozuk az ABS Y értéket   }
sub    ax,y2
jnc    @t1
neg    ax
@t1:          { Az ABS Y az AX-ben van                               }
rcr    bl,1          { Az első bit az sgn(Y)=1 feltételből adódik }

```

Ez a lépés talán magyarázatra szorul. Hogyan kerül be az *rcr bl,1* hatására az első *sgn Y=1* feltétel eredménye a BL-be? Ha az *Y1* kisebb, mint az *Y2* — tehát a vonalat felfelé kell majd húzni —, akkor az átvitel jelzőbit a kivonáskor bebillen. Ekkor negáljuk az értéket, mivel az AX-ben most egy negatív szám van, de a negálás ismét 1-be állítja az átvitelbitet. Tehát amikor a végrehajtás az *rcr bl,1* utasításhoz ér, akkor *Cy=1*, ha *Y2<Y1* és *Cy=0*, ha *Y2>=Y1*. Ezért a BL-be akkor kerül 1, ha felfelé kell lépni. Az *sgn X* meghatározása ugyanígy történik.

```

mov    si,x1         { Az SI-ben létrehozuk az ABS X értéket   }
sub    si,x2
jnc    @t2
neg    si
@t2:          { Az ABS X az SI-ben }
rcr    bl,1          { A második bit az sgn(X)=1 feltételből adódik }
cmp    si,ax
rcr    bl,1          { A harmadik bit az ABS(X)<ABS(Y) feltételből
                    származik                               }

```

A harmadik bit az *abs X-abs Y* kivonásból adódik, mivel a **cmp** utasítás valójában kivonást végez, és így előáll a hárombites érték a BL felső három bitjén. Most ezt az értéket el kell tolni, hogy indexként tudjuk használni a fenti *LrutTab* táblázathoz.


```
shr    bl,4      { Eltoljuk a megfelelő bithelyre,
                  BX = 0,2,4,6,...14 }
{ A megfelelő rutinnal kirajzoljuk a vonalat }
```

Az eltolás után a BX egy 0 és 14 közötti páros számot tartalmaz. Ha ehhez hozzáadjuk a táblázat kezdőcímét, akkor a BX máris a megfelelő rutin címére fog mutatni. E rutin meghívása után az egyenes megjelenik a képen.

```
add    bx,offset LrutTab
mov     r1,bp     { Tároljuk a BP regisztert, mert a részrutinok
                  használják majd }
call   word [bx] { Ugrás a megfelelő A..H rutinra }
mov     bp,r1     { Helyreállítjuk a BP regisztert, kilépés }
End;
```

A részrutinok nagyon hasonlítanak egymáshoz, ezért nézzük meg most példaképpen a *Line_A* rutint, amely az „A” mezőbe tartozó vonalakat húzza.

```
Procedure Line_A; Assembler; { Az "A" mezőbe tartozó vonalak
                                kirajzolása }

Asm
mov     bp,ax     { BP-be tesszük az ABS Y értéket }
mov     cx,ax     { A CX-be is ez kerül, mert ennyi pontot
                  kell kirajzolni }
mov     bx,si     { A BX-be az ABS X érték kerül }
mov     si,ax     { Az SI átveszi az ABS Y értéket }
shr     si,1      { SI = INT ( ABS Y / 2 ) }
mov     ah,LineBITpt { Az AH most a legelső pixel bitpozícióját
                  tartalmazza }

{ Az @llp ciklus végzi a vonal pontjainak kirajzolását }
@llp:
mov     al,2      { A Write Plane Enable regiszter száma az AL-be }
add     si,bx     { SI = int(abs(Y)/2) + abs(X) }
jc      @diag     { Túlcsoordulás esetén átlós lépés következik }
cmp     si,bp
jc      @vhl      { Főirányú a lépés, ha az int(abs(Y)/2) +
                  abs(X) < abs(Y) }

{ Egy átlós (az A esetben -X) lépés }
@diag:
sub     si,bp     { Korrigáljuk az SI-t a következő számításához }
ror     ah,1      { A bitmaszk jobbra (a képen a pixel balra) lép }
out     dx,ax     { A maszk kiírása a Plane Write Mask regisz-
                  terbe }
sbb     di,0      { A cím csökken, ha a rotáláskor a bit ki-
                  lépett jobbra }
```

```

    { Egy főirányú (az A esetben -Y) lépés }
@vhl:
    sub    di,Lsize    { A cím csökkentése, egy lépéssel feljebb
                        lépünk }

    { A lépés kirajzolása a képre }
@nxt:
    mov    al,LineColor { A színt kivesszük a tároló változóból }
    mov    ES:[ di ],al  { A pixelt kirakjuk a képre }
    loop   @llp          { Minden egyes pixelt kirakunk }
End;

```

9.5.2. Függőleges vonal rajzolása

A függőleges vonal rajzolásakor azt tudjuk kihasználni, hogy a vonal pixelei egymás alatt lévő képbájtokban, és azonos bithelyeken találhatók. Ezért az első pixel kirajzolása után egyetlen ciklussal ki tudjuk rajzolni az összes pontot.

Procedure Vline(X,Y,Long:word); **Assembler;**

Asm

```

    mov    cx,Long      { A rutin nem fut le, ha a vonal hosszúsága
                        nulla }

    jcxz   @exit
    push   x             { Kirajzoljuk a kezdőpontot }
    push   y
    mov    al,Ink
    push   ax
    call   PutPixel      { A kezdőpont megjelenik }

    { Az ES most már a képre mutat, a DI pedig a képbájtra }
    { A PutPixel beállította a $3D4,$02 Plane Write Mask }
    { regisztert is, }
    { az AL-ben pedig az INK szín van }

    dec    cx
    jz     @exit         { Kiugrik, ha a hossz 1, mert akkor már
                        készen van }

@vln:
    add    di,Lsize      { A DI-t a következő képbájtra állítjuk }
    mov    ES:[ di ],al  { A pontok kirajzolása egymás alá }
    loop   @vln

@exit:
End;

```

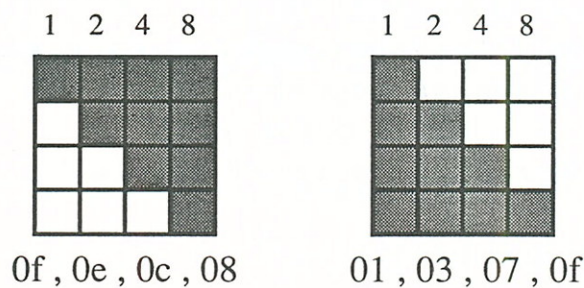

9.5.3. Vízszintes vonal rajzolása

A vízszintes vonal hosszától függő eseteket már megismertük a 16 színű *Hline* eljárásnál. Itt ugyanazt az elvet tudjuk alkalmazni:

1. Ha a végpont és a kezdőpont ugyanabban a képbájtban helyezkedik el, akkor csak egyetlen bájtot kell módosítani.
2. Ha a végpont és a kezdőpont nem ugyanabban a képbájtban helyezkedik el, akkor kell egy első bájt maszk és egy utolsó bájt maszk, és a köztes bájtok minden bitjét *INK* színűre kell színezní.

A 16 színű módszert módosítanunk kell, mert itt egy képcímen csak 4 pixel található, ezért az első és az utolsó maszkok (*Fmask* és *Lmask* táblázat) másfajta:

Fmask:



48. ábra Az első és utolsó bájt maszkok a vízszintes vonalakhoz

Hogy a fenti két eset közül melyik teljesül, azt az $X \text{ shr } 2$, azaz a kezdőpont bájt pozíciója és az $(X+Long) \text{ shr } 2$, azaz a végpont bájt pozíciója határozza meg.

$X \text{ shr } 2 = (X+Long) \text{ shr } 2$: egyetlen bájt

$X \text{ shr } 2 < (X+Long) \text{ shr } 2$: két szélső bájt és köztes bájtok

Procedure Hline(X,Y,Long:word); **Assembler;**

Asm

```

mov    cx,Long
jcxz   @exit      { Nincs vonalhúzás, ha a hosszúság nulla }

cld
mov    ax,Lsize   { Címszámítás }
mul    Y
mov    di,X
shr    di,2
add    di,ax      { A képcím a DI-ben van }
mov    dx,03C4h
mov    ES,SegA000
```

```

{ ES = SegA000  DX = 03c4h, DI = képcím }

mov     ax,X
mov     si,ax      { SI = kezdő X koordináta }
add     cx,si      { CX = kezdő X koordináta + hosszúság }
dec     cx         { A CX a végpont X koordinátáját tartalmazza }
lea     bx,Fmask   { Elsőként az első bájt maszkot határozzuk meg }
and     al,3       { Az X alsó két bitje adja a táblázat indexét }
xlat    { Kivesszük a megfelelő bitmaszkot }
mov     ah,al      { Az AH-ba teszi az első bájt maszkot }
lea     bx,Lmask   { Most meghatározzuk az utolsó bájt maszkot }
mov     al,cl      { Az utolsó pont X koordinátájának alsó bájtja }
and     al,3       { Csak az alsó két bitre van szükség }
xlat    { Az AL-be teszi az utolsó bájt maszkot }
mov     bl,Ink     { A rajzolás INK színnel fog történni,
                    ez a BL-ben lesz }
shr     cx,2       { Meghatározzuk a bájthatárokat }
shr     si,2       { Az alsó két bit nem kell }
sub     cx,si      { Bájthatár-ellenőrzés és a közbenső bájtok
                    { számának meghatározása a CX-ben
                    (CX-1 darab van) }
jz      @one_byte { Ugrás, ha a vonal egy bájton belül van }

{ A vonal legalább két bájtból áll }

mov     bh,al      { Az utolsó maszk tárolása későbbre }
mov     al,02h
out     dx,ax      { Az első maszk kiírása }
mov     ES:[di],bl { A vonal első bájtját kitesszük a képre }
inc     di
dec     cx
jz      @last      { Ugrás, ha nincsenek közbenső bájtok }

{ Ha vannak közbenső bájtok, akkor azoknak 1111b a maszkja ! }

mov     ah,0fh     { A közbenső bájtok maszkja 1111b }
out     dx,ax
mov     al,bl      { Az INK szín az AL-be kerül }
rep     stosb      { A közbenső bájtok kirakása }
@last:
mov     ah,bh      { Az utolsó maszk kiírása }
mov     al,02h
out     dx,ax
mov     ES:[di],bl { A vonal jobb szélének kirakása
                    (a vonal vége) }
jmp     @exit      { Kilépés }

{ A vonal elfér egy bájtban }

```



```

@one_byte:
    and    ah,al          { Ilyenkor a két maszk AND kapcsolata kell    }
    mov    al,02h
    out    dx,ax          { A maszk kiírása                                }
    mov    ES:[di],bl     { A bájt kirakása                                }
@exit:
    End;

```

9.6. Téglalapok rajzolása

9.6.1. Üres téglalap rajzolása

Az üres téglalapot az előzőekben ismertetett *Vline* és *Hline* eljárások felhasználásával két-két párhuzamos vonal húzásával rajzoljuk ki. Mivel az eljárás igen egyszerű, és az előző fejezetekben is volt már rá példa, ezért a működésének ismertetésétől itt eltekintünk.

```
Procedure Box(X1,Y1,X2,Y2:word); Assembler;
```

9.6.2. Kitöltött téglalap rajzolása

A kitöltött téglalapot hasonlóan rajzoljuk, mint a 16 színű grafika esetén. A *Hline*-ban leírtak alapján meghatározzuk a vízszintes első és utolsó bájt maszkot, a közbenső bájtokat, valamint a téglalap magasságát. Elsőként kirajzoljuk az első bájtokat, ez egy függőleges oszlop, majd a közbenső téglalapot (ha vannak közbenső bájtok), végül az utolsó bájtokat. Ha a téglalap vízszintesen elfér egy képbájtban, akkor ezt külön ciklussal rajzoljuk ki.

```
Procedure Bar(X1,Y1,X2,Y2:word); Assembler;
Asm
```

Első lépésként a koordinátákból meg kell határozni a bal felső sarkot, annak memóriacímét, valamint a téglalap hosszúságát és magasságát.

```

mov    cx,X1
mov    dx,Y1          { Az X1,Y1-et feltételezi bal felső saroknak }
{ Az X koordináták sorrendjét nézi meg, és megcseréli őket,
    ha kell
}
mov    ax,X2          { A vízszintes hosszúság megállapítása      }
sub    ax,X1

```

```

    jnc     @t1
    xchg    cx,X2          { Koordinátacsere, ha szükséges }
    neg     ax
@t1:
    inc     ax             { AX a vízszintes hosszúság }
    { Az Y koordináták sorrendjét nézi meg, és megcseréli őket, ha }
    { kell }
    mov     bx,Y2          { A függőleges hosszúság megállapítása }
    sub     bx,Y1
    jnc     @t2
    xchg    dx,Y2          { Koordinátacsere, ha szükséges }
    neg     bx
@t2:
    inc     bx             { BX a függőleges hosszúság }
    { Tároljuk a hosszúságokat }
    mov     r1,ax           { r1 = AX, tárolás későbbre }
    mov     r2,bx           { r2 = BX, tárolás későbbre }
    }

    mov     r3,cx           { A kezdőkoordinátát is tároljuk r3 = X1 }
    mov     r4,dx           { r4 = Y1 }
    { Az r3,r4 illetve az X2,Y2 most már valóban a helyes vég- }
    { pontokat adja }
    mov     ax,Lsize        { Most kiszámítjuk a bal felső sarok kép- }
    { címét }
    mul     dx
    mov     di,cx
    shr     di,2
    add     di,ax           { A DI-ben most a képcím van }
    mov     ES,SegA000      { Az ES mutat a képre, }
    mov     dx,03c4h        { a DX pedig a Write Plane Enable regiszterre }
    { Most a Hline mintájára meg kell határozni a maszkokat }
    mov     al,cl
    mov     si,cx           { SI = kezdő X koordináta }
    add     cx,r1           { CX = végső X koordináta }
    dec     cx
    lea     bx,Fmask        { Az első bájt maszk táblázat }
    and     al,3            { Csak az alsó két bit szükséges }
    xlat                     { Kivesszük a megfelelő elemet }
    mov     ah,al           { Az AH-ba teszi az első bájt maszkot }
    lea     bx,Lmask        { Az utolsó bájt maszk táblázat }
    mov     al,cl           { Az utolsó pont X koordinátájának alsó }
    { bájtja }
    and     al,3            { Csak az alsó két bit szükséges }
    xlat                     { Az AL-be teszi az utolsó bájt maszkot }
    mov     bl,Ink          { A rajzolás INK színnel fog történni }
    mov     r0,ax           { A maszkokat tároljuk későbbre }
    shr     cx,2

```



```

shr    si,2          { Az alsó két bit nem kell }
sub    cx,si         { Bájthatár-ellenőrzés }
jz     @one_byte     { Elugrik, ha a BAR függőlegesen belefér
                      egy bájtba }

mov     r6,cx         { Vízszintes hosszúság az r6-ba }
mov     r5,di         { A bal felső sarok címe az r5-be }
mov     r7,di         { és az r7-be is }
inc     r5            { Növeljük a címet (későbbi használatához) }

mov     al,02h        { Az első bájtok maszkja }
out     dx,ax         { kikerül a Write Plane Enable regiszterbe }
mov     cx,r2         { CX-be tesszük a magasságot }

@first_bytes:
mov     ES:[di],bl     { Egy bájt kiküldése a képernyőre }
add     di,Lsize
loop   @first_bytes   { A ciklus kirajzolja a téglalap bal szélét }

dec     r6            { A közbelső kitöltött bájtok meghatározása }
jz     @lasts         { Ugrás, ha nincsenek közbelső bájtok }
mov     si,r2         { Az SI a függőleges hosszúságot adja }
mov     ah,0fh        { A közbelső bájtok maszkja 1111b }
out     dx,ax
mov     al,bl         { Az INK szín az AL-be }

@middle_bytes:
mov     cx,r6         { A vízszintes hosszúság a CX-be }
mov     di,r5         { A képcím a DI-be kerül }
rep     stosb         { Egy vízszintes sor feltöltése }
add     r5,Lsize      { A cím növelése }
dec     si
jnz     @middle_bytes { Minden közbelső sort kirajzolunk }
mov     di,r6         { A DI-be a hossz }
add     r7,di         { Hozzáadjuk az eredeti címet }
{ Most tehát az r7 = jobb felső sarok címe-1 }

@lasts:
{ A jobb szélső bájtok kirajzolása következik }
mov     di,r7         { DI-be a cím }
inc     di            { Egygel arrébb kell léptetni }
mov     al,02h        { Az utolsó bájt maszk }
out     dx,ax         { kivitele a Write Plane Enable regiszterbe }
mov     cx,r2         { A magasság a CX-be kerül }

@last_bytes:
mov     ES:[di],bl     { Egy bájt kiküldése a képre }
add     di,Lsize
loop   @last_bytes    { A ciklus kirajzolja a téglalap jobb szélét }
jmp     @exit         { Kilépés }

{ Ez a rész akkor fut le, ha a téglalap vízszintesen elfér egy
  bájtban }

```

```

@one_byte:
    and    ah,al           { A két maszkot AND kapcsolatba kell hozni }
    mov    al,02h
    out    dx,ax
    mov    cx,r2
@one_bytes:
    mov    ES:[di],bl      { A függőleges BAR felrajzolása }
    add    di,Lsize
    loop   @one_bytes
@exit:
    End;

```

9.6.3. Keretezett téglalapok

A keretezett téglalapokról is volt már szó. A *Bbar* és a *Cbar* eljárások egyszerűen a *Bar*, a *Vline* és a *Hline* rutinok hívásával állítják elő a térhatású téglalapokat. A két program szinte teljesen azonos a *normál bájt/pixel* képformátumnál ismertetett rutinokkal, ezért felesleges lenne újra leírni őket, hiszen csak a fenti eljárások hívásaiból áll mindkettő.

```

Procedure Bbar(X1,Y1,X2,Y2,Ty:word); Assembler;
    { Vastag keretű téglalap }
Procedure Cbar(X1,Y1,X2,Y2,Ty:word); Assembler;
    { Vékony keretű téglalap }

```

9.7. Képterületek tárolása és másolása

A képterületek elmentése ebben a képszerkezetben sajnos gondot okoz. A legnagyobb probléma az, hogy az *X* irányban nehéz egy területet pixelhatáron kezdve kijelölni. Amint azt a *Hline* és a *Bar* eljárásokban láthattuk, a két végpont miatt az első és az utolsó képbájtot vízszintesen maszkolni kell. Az elmentésnél ez ugyanígy van. Két lehetőség közül választhatunk:

1. Megoldjuk a vízszintes irányú pixeles tárolást, ill. koordinátamegadást, még akkor is, ha ez eléggé hosszadalmas és bonyolult algoritmust igényel.
2. Vízszintes irányban csak bájt határon, azaz négy pixelenként tesszük lehetővé a koordináták megadását, és a tárolás is mindig ebben a formában történik. Ezzel a megkötéssel a területeket ugyan nem tudjuk majd pixelhatáron tárolni vagy megjeleníteni, de az előző esethez képest lényegesen egyszerűbb és persze gyorsabb eljárást kapunk.

...és mi természetesen a második, az egyszerűbb megoldást választjuk!

A következő *GetImage* eljárás elmenti az $X1, Y1$ és $X2, Y2$ koordinátákkal megadott ablakot a *Dest* típus nélküli változó által meghatározott területre. A koordinátákra szigorú megkötések vannak. Az $X1$ és az $X2$ (négypixeles) bájtathárokat jelölnek, tehát az értékeiknek a $0..79$ intervallumba kell esniük, és teljesíteni kell az $X2 \geq X1$ és $Y2 \geq Y1$ feltételeket. Például a kép bal felső sarkában lévő 32×32 -es ablakot a következő paraméterekkel lehet elmenteni:

```
GetImage(0, 0, 7, 31, Dest);
```

(Az eljárás hívása előtt az *ImageSize* függvénnyel meg kell határozni az ablak méretét, és le kell foglalni számára a megfelelő méretű memóriát.)

Procedure GetImage($X1, Y1, X2, Y2$:word; **var** Dest); **Assembler;**

Asm

```
push    DS
mov     DS, SegA000      { A DS a képernyőre mutat, itt lesz a
                          forráscím }
cld
les     di, Dest         { ES:DI mutat a célcímre }
```

Első lépésként a koordináták alapján meghatározzuk a képcímet és az ablak méreteit.

```
mov     ax, X2            { A szélesség meghatározása }
sub     ax, X1
inc     ax
stosw   { A szélesség az első adat, amelyet
        tárolni kell }
mov     bx, ax            { A BX-ben is tároljuk }

mov     ax, Lsize         { A képcím meghatározása }
mul     Y1
add     X1, ax            { A címet az X1-ben tároljuk }

mov     ax, Y2            { A magasság meghatározása }
sub     ax, Y1
inc     ax
stosw   { A magasság a második adat }
mov     Y1, ax            { Az Y1-ben tároljuk }
```

A képcím és a méretek meghatározása után pixelenként kimásoljuk a képet. A másolás úgy történik, hogy elsőként kimásoljuk az összes képbájt 3. bitplane-jén található bájtot, majd a 2. bitplane minden pixeljét és így tovább, mind a négy bitplane-t. Ez azt eredményezi, hogy egy bájt és az azt követő bájt az eltárolt memóriatömbben nem két egymás mellett levő pixelt tartalmaz, hanem minden negyedik pixelt (fésült tárolás).

```

    mov     ax,0304h                { Read Plane Mask regiszter      }
@cikl1: { A bitplane-eket elmentő ciklus}
    mov     dx,03ceh
    out     dx,ax                   { A következő olvasandó plane
                                   kiválasztása                       }
    mov     dx,Y1                   { DX számolja a sorokat          }
    mov     si,X1                   { SI-be a kezdőcím                }
@cikl2: { Az egy bitplane-t elmentő ciklus }
    push    si
    mov     cx,bx                   { BX darab bájt van egy sorban    }
    rep     movsb                   { Kimásolja a sort                }
    pop     si
    add     si,Lsize                { Címnövelés, a következő sor    }
    dec     dx
    jnz     @cikl2                  { Minden sort kimásol            }
    dec     ah                      { A plane-szám csökkentése        }
    jns     @cikl1                  { Minden plane-t kimásol          }

    pop     DS
End;
```

A következő *PutImage* az előző módszerrel elmentett képet visszamásolja a képernyő megadott *X,Y* pozíciójába, ahol az *X* képbájt pozíció (0..79), míg az *Y* természetesen a rasztensor szám (pixelpozíció).

Procedure PutImage(*X,Y:word; var Source*); **Assembler;**

Asm

```

    push    DS
    mov     ES,SegA000 { Az ES rámutat a képre, az ES:DI most a célcím}
    cld
    lds     si,Source   { DS:SI mutat a tárolt képre, ez a forráscím  }
```

Első lépésként kiszámoljuk az *X,Y* koordinátákhoz tartozó képcímet.

```

    mov     ax,Lsize        { Címszámítás a koordinátákból      }
    mul     Y
    add     X,ax            { A képcímet az X paraméter tárolja }
```

Beolvassuk a tárolt adatok első két word adatát, ez lesz a szélesség (képbájtokban) és a magasság (pixelesen).

```

    lodsw                { Az első adat a szélesség }
    mov     bx,ax        { A szélesség a BX-be kerül }
    lodsw                { A második adat a magasság }
    mov     Y,ax         { A magasság az Y-ba kerül }
```


Most az elmentéshez hasonlóan, de fordított irányban a memóriából sorra visszamásoljuk minden bitplane tartalmát. A bitplane-ek kiválasztását a \$3C4, \$02 *Write Plane Enable* regiszter végzi.

```

    mov     ax,0802h                { A Write Plane Enable regiszter }
@cikl1: { Ez a ciklus minden bitplane-t visszamásol }
    mov     dx,03c4h
    out     dx,ax                   { A plane-maszk kiírása a regiszterbe }

    mov     dx,Y                    { Ennyi sorból áll a kép }
    mov     di,X                    { DI-be tesszük a címet }
@cikl2: { Az egy bitplane-t visszamásoló ciklus }
    push    di
    mov     cx,bx                   { Egy sor BX számú bájtból áll }
    rep     movsb                   { Kimásol egy sort a képre }
    pop     di
    add     di,Lsize                { Címnövelés, a következő sor }
    dec     dx
    jnz     @cikl2                  { Minden sort kitesz }
    shr     ah,1                    { A következő plane-re mutat }
    jnc     @cikl1                  { Minden plane-t kitesz }

    pop     DS
End;
```

Az *ImageSize* függvény meghatározza egy ablak méretét. A koordinátákra ugyanaz érvényes, mint a *GetImage*-nél. Ha a megadott terület mérete nagyobb 64K-nál, akkor a függvény nulla értékkel tér vissza. Túl nagy területeket kisebb részekre bontva lehet elmenteni.

Function ImageSize(X1,Y1,X2,Y2:word):word; **Assembler;**

Asm

```

    mov     ax,X2                    { A szélesség meghatározása }
    sub     ax,X1
    inc     ax                        { Vízszintes méret az AX-be }
    mov     dx,Y2                    { A magasság meghatározása }
    sub     dx,Y1
    inc     dx                        { Függőleges méret a DX-be }
    mul     dx                        { Összeszorzás (ez csak a terület negyede) }
```

Mivel a legnagyobb vízszintes méret 80, a legnagyobb függőleges pedig 480, ezért a szorzat eredménye itt még legfeljebb 38 400 lehet, ha a paramétermegadás nem hibás. A szorzás csak a terület negyedét adja meg, hiszen egy képbájton négy pixelt tárolunk, ezért az AX-et még meg kell szorozni négygyel.

```

shl    ax,1                { Szorzás 2-vel }
jc     @overflow           { Ha túlcsordul, akkor a terület túl nagy }
shl    ax,1                { Szorzás ismét kettővel }
jc     @overflow           { Ha túlcsordul, akkor a terület túl nagy }
add    ax,4                { +4 a hosszúságok tárolása miatt }
jnc    @exit               { Ha nem csordul túl, akkor a terület
                           megfelelő }

@overflow:
xor     ax,ax              { Túl nagy területek esetén }
                           { a visszatérési érték 0 }

@exit:
End;

```

9.8. Karakter és szöveg megjelenítése

A karaktereket a *normál bájt/pixel* módhoz hasonlóan itt is pixelesen írjuk ki, de a képszerkezet miatt a megoldás kissé bonyolultabb lesz. Egy karakter kiírását a *Char8x8* eljárás végzi, és ezt hívjuk a későbbi *OutCharXY* és *OutStrXY* eljárásból. A *Char8x8* három bemenő paramétert igényel:

- a képcímet a *Cadr* változóban,
- a képcímen belüli bitmaszkot a *Cbit* változóban,
- a karakter kódját az *AL* regiszterben.

A kiírás *Ink* színnel történik, az *Over* változó vezérli, hogy a háttér *Paper* színű legyen-e, vagy változatlan maradjon. Mivel a rutin ezenkívül még néhány inicializálást igényel, ezért önmagában nem működőképes.

A karakter kirajzolásának folyamata a következő: a kód alapján meghatározzuk a *charset* karakterkészletben található 8-bájtos bittérkép címét, és betöltjük azt az *SI* regiszterbe. Ezt követően bitenként haladva kirajzolunk minden egyes pixelt a képre. A képszerkezet miatt ez egy kissé hosszadalmas eljárás, de még így is megfelelően gyors.

Procedure Char8x8; **Assembler;**

Asm

{ Előzetes beállítások szükségesek :

ES = \$A000, a kép szegmenscíme

DX = \$03C4, sequencer regiszterek }

mov bh,Ink { Az írás színét a BH regiszter adja majd }

Az első lépés a kód alapján a karakterminta kezdőcímének meghatározása.

```
xor  ah,ah      { A kód alapján meghatározzuk a karakter címét}
shl  ax,3       { A kód megszorozása 8-cal (AX eltolása)      }
lea  si,Charset { A karakterkészlet kezdőcíme                }
add  si,ax      { A CS:SI most a megfelelő karakterre mutat }
```

Megcímezzük az első, bal felső sarok pixeljét, és a képbájtban kimaszkoljuk azt a \$3C4, \$02 regiszterrel.

```
mov  di,Cadr     { Képcím a DI-be                               }
mov  ah,Cbit     { Bitpozíció az AH-ba                         }
mov  al,2        { A Write Plane Enable regisztert használjuk }
out  dx,ax       { Az első pixel pozícióját kivisszük a regisz- }
                  terbe                                         }
mov  ch,8        { A CH számlálja a karakter bájtjait         }
```

Az *Over* értékétől függően a rutin most két részre ágazik. Az egyik esetben a háttér színű pixelek (amely a bitmintában 0) változatlanok maradnak, a másik esetben pedig *Paper* színűek lesznek.

```
cmp  Over,0      { A háttérre rajzolást az Over határozza meg }
jz   @paper      { Over = 0 esetben a Paper színt viszi a hát- }
                  térre                                         }
```

```
{ Nincs háttérmódosítás, ha az Over<>0 }
```

A külső *@cikl0* ciklus a nyolc bájtot veszi sorra.

```
@cikl0:
mov  bl,CS:[si]  { Egy bájt behívása a karaktermintából       }
inc  si          { A forráscím növelése                       }
push di         { A kezdőcímet és a bitpozíciót a STACK-ben   }
                  tároljuk                                     }
push ax
```

A belső *@cikl1* ciklus egy bájt nyolc bitjét veszi sorra.

```
mov  cl,8        { Egy vízszintes sor nyolc bitből áll       }
@cikl1:
shl  bl,1        { A következő bit a Carry-be kerül         }
jnc  @t1         { Ugrás, ha háttér színű, ekkor nincs      }
                  kiírás                                       }
mov  ES:[di],bh  { Egy pixel kimegy az Ink színnel, ha a Carry=1 }
```

```

@t1:
    rol    ah,1        { A képpontpozíció léptetése }
    out    dx,ax        { Plane maszk ($3C4, $02) }
    adc    di,0        { A cím is nő, ha a bit kilépett a négybites
                        határból }
    dec    cl          { Csökkentjük a bitszámlálót }
    jnz    @cikl1       { Az összes bit kirakása }
    pop    ax          { Visszahozzuk az eredeti címet és bitpozíciót }
    pop    di
    out    dx,ax        { Újra kivisszük a bal oldali első pixel
                        pozícióját }
    add    di,Lsize     { A képcím növelése, a következő sor }
    dec    ch          { Csökkentjük a bájt számlálót }
    jnz    @cikl0       { Az összes bájt kirakása }
    jmp    @exit        { Átugorjuk a Paper esetet, kilépés }

@paper:
    { Van háttérmódosítás, ha az Over=0 }

@cikl2:
    mov    bl,CS:[si]   { Egy bájt behívása a karaktermintából }
    inc    si           { A forráscím növelése }
    push   di           { A kezdőcímet és a bitpozíciót a STACK-ben
                        tároljuk }
    push   ax
    mov    cl,8         { A CL számlálja a biteket (vízszintes méret) }

@cikl3:
    shl    bl,1         { A következő bit a Cy-ba kerül }
    jc     @t2          { Ha Cy=1, akkor Ink szín kell }
    mov    bh,Paper     { Ha Cy=0, akkor pedig Paper }
    mov    ES:[di],bh   { A Paper színű pixel kirakása a képre }
    mov    bh,Ink       { A BH ismét az Ink szint tartalmazza }
    jmp    @t3          { Az Ink kirajzolás átugrása }

@t2:
    mov    ES:[di],bh   { Egy pixel kimegy Ink színnel }

@t3:
    rol    ah,1         { A következő pixel bitpozíciója }
    out    dx,ax        { A Write Plane Enable ($3C4, $02) regiszterbe
                        megy }
    adc    di,0        { A cím is nő, ha a bit kilépett a négybites
                        határból }
    dec    cl          { A bitszámláló csökkentése }
    jnz    @cikl3       { Az összes bit kirakása }
    pop    ax          { Az eredeti cím és bitpozíció visszatöltése }
    pop    di
    out    dx,ax        { Az első pozíciót újra kivisszük }
    add    di,Lsize     { A képcím növelése, a következő sor }
    dec    ch          { Csökkentjük a bájt számlálót }
    jnz    @cikl2       { Az összes bájt kirakása }

@exit:
    add    Cadr,2       { A címet kettővel (azaz 2*4 pixellel) növeljük }
    End;

```


A most következő *OutCharXY* eljárás kiír egy karaktert. Az *X,Y* koordináták a karakter bal felső sarkának pozícióját adják meg, a *C* pedig a karaktert. A kiírást az előző *Char8x8* rutin hívásával végezzük el a képcím kiszámítása és az inicializálások után.

Procedure OutCharXY(*X,Y:word;C:char*); **Assembler;**

Asm

```

mov  di,X          { Kiszámítjuk az X,Y által meghatározott
                    képcímet }
mov  ax,Lsize
mul  Y              { Y* Lsize }
shr  di,2           { X shr 2 }
add  ax,di          { Az AX-ben megjelenik a képcím }
mov  Cadr,ax        { A kiszámított címet a Cadr-ba teszi }
mov  ax,X           { Meghatározzuk a bitpozíciót az X AND 3
                    művelettel) }
and  al,3           { Az alsó kétbites szám lesz a táblázat indexe }
lea  bx,Pixelpos    { A PixelPos táblázat tartalmazza a bit-
                    maszkokat }
xlat                { A táblázat megfelelő elemét kivesszük az
                    AL-be }
mov  Cbit,al        { A bitmaszkot a Cbit változóba tesszük }
mov  al,C           { A karakter kódja az AL-be kerül }
mov  dx,03c4h       { A DX-be a sequencer regiszterek címe kerül }
cld
mov  ES,SegA000     { Az ES a képre mutat }
call Char8x8        { Kiírja a karaktert }

```

End;

Az *OutStrXY* egy karakterlánc kiírását végzi. Az eljárás meghatározza a képcímet és a bitpozíciót, majd a *Char8x8* rutinnal karakterenként kiírjuk a szöveget.

Procedure OutStrXY(*X,Y:word; S:string*); **Assembler;**

Asm

```

mov  ES,SegA000     { AZ ES-t ráállítjuk a képre }
mov  di,X           { Képcímszámítás a koordináták alapján }
mov  ax,Lsize
mul  Y              { Y*Lsize }
shr  di,2           { X shr 2 }
add  ax,di          { Az AX-ben megjelenik a képcím }
mov  Cadr,ax        { A kiszámított címet a Cadr-ba teszi }
mov  ax,X           { Meghatározzuk a bitpozíciót }
and  al,3
lea  bx,Pixelpos    { A bitpozícióból meghatározzuk a bitmaszkot }
xlat
mov  Cbit,al        { A bitmaszk a Cbit változóba kerül }
cld
mov  dx,DS           { A DX az eredeti adatszegmenset tárolja }

```

```

lds    si,S           { A DS:SI a sztringre mutat }
mov    bx,DS          { A BX a sztring szegmensét tárolja }
lodsb                     { A sztring első bájtja a sztring hosszúságát
                           adja }

mov    cl,al
xor    ch,ch          { A hosszúság a CX-be kerül }
jcxz   @exit          { Kiugrás, ha a CX nulla }

{ A @cikl ciklus sorra kiírja a karaktereket }
@cikl:
push   cx             { Tárolja a számlálót }
mov    DS,bx
lodsb                     { Veszi a következő karaktert }
push   si             { Tárolja a forráscímet }
mov    DS,dx           { A DS megkapja az eredeti értékét }
push   dx             { Tároljuk az eredeti DS-t }
push   bx             { Tároljuk a sztring szegmensét }
mov    dx,03c4h        { A Write Plane Enable regisztert használjuk }
call   Char8x8         { Kiírja a karaktert a Char8x8 rutinnal }
pop    bx             { Az elmentett adatokat visszahozzuk }
pop    dx
pop    si
pop    cx
loop   @cikl          { Minden karaktert kiír a képre }
@exit:
mov    DS,dx          { A DS visszakapja az eredeti értékét }
End;

```

9.9. Paletta beállítása és lekérdezése

A paletta kezeléséről már szóltunk, de a teljesség kedvéért ebben az egységben is elhelyeztük a két alapvető beállító és lekérdező rutint. A *SetRGBreg*-ben a *regnum* adja a beállítandó regiszter számát, a *Red*, *Green*, *Blue* adatok pedig az alapszín komponenseket. A *GetRGBreg* esetén a *regnum* számú RGB regisztert kérdezzük le, és a hárombájtos adattömböt az *RGB* típus nélküli változóba tesszük le. Az *RGB* lehet egy tömb címe vagy bármilyen 3 bájt hosszú adattípus.

Procedure SetRGBreg(regnum,Red,Green,Blue:byte); **Assembler;**

Asm

```

mov    dx,03c8h        { RGB írás címregiszter }
mov    al,regnum
out    dx,al           { A regiszterszám megadása }
inc    dx              { $3C9 az RGB adatregiszter }
mov    al,Red          { Piros komponens }

```



```

    out  dx,al
    mov  al,Green      { Zöld komponens }
    out  dx,al
    mov  al,Blue       { Kék komponens }
    out  dx,al
End;

Procedure GetRGBReg (Regnum:byte; var RGB); Assembler;
Asm
    cld
    les  di,RGB         { ES:DI mutat a célcímre }
    mov  dx,03c7h       { Az RGB regiszterek olvasása }
    mov  al,Regnum
    out  dx,al          { A megadott regiszter kiválasztása olvasásra }
    add  dx,2           { DX=$3C9 }
    in   al,dx
    stosb               { A piros komponens lekérdezése és tárolása }
    in   al,dx
    stosb               { A zöld komponens lekérdezése és tárolása }
    in   al,dx
    stosb               { A kék komponens lekérdezése és tárolása }
End;

```

9.10. A képmemória törlése

A képmemória törlése általában azt jelenti, hogy minden képbájtot kinullázunk, azaz a képet háttér színűre váltjuk. Az alábbi rutin a teljes 64K-s videomemóriát és annak minden bitplane-jét kinullázza. Arra figyelni kell, hogy a \$3CE, \$08 bitmaszk regiszter hatással van a memóriába írásra, ezért ebben \$ff-nek kell lennie, máskülönben a törlés nem tökéletes. Az **GR256_2** egységben nincs olyan eljárás, amely módosítaná ezt a regisztert, de ha más programok hívásából ez a rutin nem törölné teljesen a képet, akkor ki kell egészíteni a bitmaszk regiszter \$ff-be állításával! A rutin egyébként az egység legelején található, mert a videomód-beállító eljárások is használják azt.

```

Procedure ClearSegA000; Assembler;
Asm
    mov  dx,03c4h
    mov  ax,0f02h       { Minden bitplane-t írásra engedélyezünk }
    out  dx,ax
    xor  ax,ax          { Nullákat kell kivinni a memóriába }
    mov  di,ax
    mov  cx,32768       { 32 768 word, azaz 65 536 bájt kinullázása }
    mov  ES,SegA000     { ES:DI mutatja a célcímet }
    cld
    rep  stosw          { Kinullázzuk a videomemóriát }
End;

```

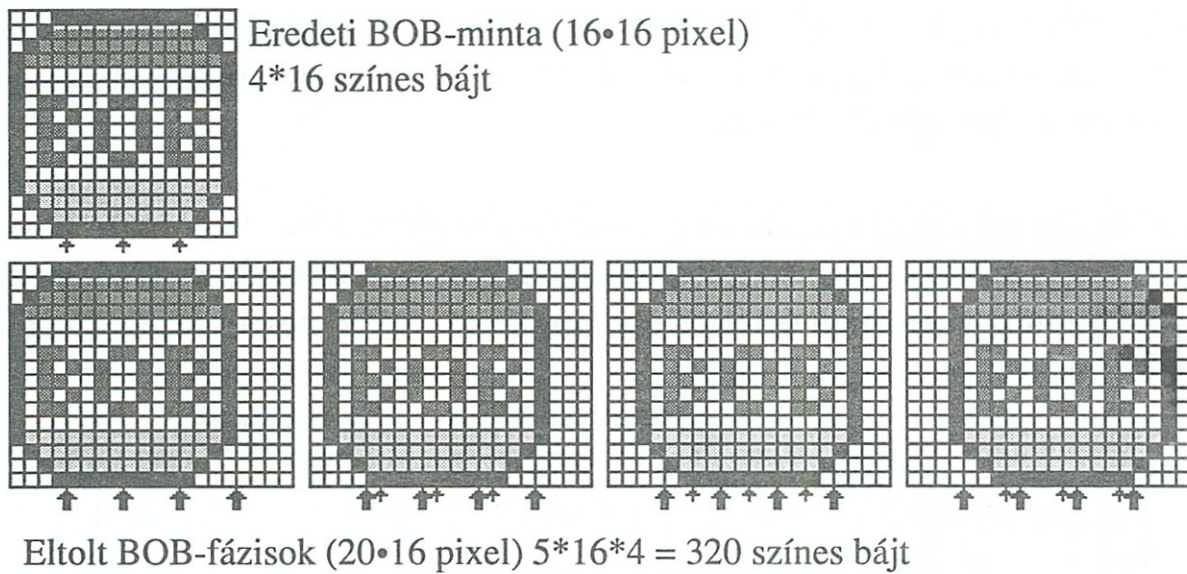

9.11. BOB-ok megjelenítése a kettősbuffer-módszerrel

A BOB-ok programozására a *módosított bájt/pixel* szerkezetű kép a legmegfelelőbb. Számos olyan tulajdonsága van, amelyek miatt előszeretettel alkalmazzák:

- A címterület lecsökkenése miatt lehetőség van a kettősbuffer-technikára.
- A BOB-ok 256 színűek lehetnek.
- A BOB-okat a képmemóriában tudjuk tárolni, mert lehetőség van a színes bájtok összemontírozására.
- A BOB-ok kirajzolása a képre leegyszerűsödik.

Az előnyök mellett meg kell említeni egy hátrányos tulajdonságot is, nevezetesen, hogy a címezhető videomemória túl kevés. Vegyünk egy egyszerű példát: osszuk fel a képmemóriát a kettősbuffer-technika használatához! Alkalmazzunk 320×240-es felbontást a képen, de ebből csak a felső 320×200-as terület legyen az aktív BOB-terület, az alsó 320×40-es területet hagyjuk meg kijelzésekre! Ekkor az alsó kijelző terület memóriaigénye $320/4 \times 40$ bájt, azaz 3200 színes bájt. A felső kép $320/4 \times 200 = 16\,000$ bájt képmemóriát igényel. Ha a kettősbuffer-módszert használjuk, akkor két munkakép szükséges (az **A** és a **B** kép), és még egy bufferkép az eredeti kép tárolásához (a **C** kép). Tehát csak a képterületek memóriaigénye már $3 \times 16\,000 + 3200$ bájt képmemória, azaz 51 200 bájt. Így a BOB-ok tárolására 14 336 színes bájt marad a 64K videomemóriából. Ha az előző BOB-programokhoz hasonlóan itt is 16×16-os BOB-okat használunk, akkor egyetlen BOB-ot (mivel 4 pixel fér el egy képbájtban) 4×16 , azaz 64 színes bájton tudunk eltárolni. De ha színes bájtokon tároljuk a BOB-okat, akkor előre el kell őket forgatni a bitpozíciók miatt, ami itt most plusz három BOB-ot jelent, mivel a képbájtok négybitesek. A forgatás miatt minden BOB vízszintes mérete megnő 1 képbájtal, tehát egyetlen BOB egy pixelpozíciójának eltárolásához $5 \times 16 = 80$ bájt kell, míg egy teljes BOB eltárolásához (4 bitpozíció van) $4 \times 80 = 320$ bájt képmemória kell. Tehát ebben az esetben a képmemóriában összesen $14\,336/320 = 44,8$ BOB fér el. Ha figyelembe vesszük, hogy a BOB-ok általában több animációs fázist tartalmaznak, akkor ez a terület bizony kevés. Egy négyfázisú mozgás esetén 11 BOB-ot tudunk tárolni. A teljes módszert a 49. ábra mutatja.

A BOB-okat a beolvasás után tehát át kell másolni a videomemóriába az ábra szerinti formában. Az átmásolást a példaprogramban a *Convert_Datas* eljárás végzi majd, amely eléggé összetett és hosszú, de elvégzi a *normál bájt/pixel* formátumról való konvertálást és a bitpozíciók miatti eltolásokat.



49. ábra A BOB-ok eltolása bitpozíciókba

Meg kell még vizsgálnunk azt is, hogyan lehet összemontírozni két színes bájtot. Legyen az összemontírozás maszkja 1001b, valamint jelöljük S -sel a forrás színes bájtot, valamint D -vel a cél színes bájtot. A feladat a következő: másoljuk át úgy az S -et a D -be, hogy a maszk 1-es bitjei által megadott helyeken az S bitjei jelenjenek meg, a 0 értékű bithelyeken pedig maradjanak meg a D bitjei!

A feladat megoldásához a következőket kell tenni:

1. A $\$3CE$, $\$08$ *bitmaszk* regisztert nullázni kell, mert ekkor minden írás mód másolást hajt végre. (Emlékezzünk vissza, hogy egy képbájt írásakor oda kerülnek a latch-ben tárolt értékek, ahol a bitmaszk regiszterben nulla van. Ha a bitmaszk 00000000b, akkor minden bitbe a latch elemei kerülnek.)
2. Be kell olvasni az S -et egy képmemória olvasással, ez a latch feltöltődését eredményezi.
3. A maszkot (amely most 1001b) ki kell vinni a $\$3C4$, $\$02$ *Write Plane Enable* regiszterbe; ezáltal megadjuk, hogy a képbájt mely bitjei módosulhatnak.
4. Egy memóriába írással a latch tartalmát ki kell vinni a D -be.

A négy lépés elvégzése után megkapjuk az összemontírozott színes bájtot. Vigyázni kell azonban, hogy ne keverjük a fogalmakat! A bitmaszk regiszter, amely alapesetben a képbájt pixeljeit maszkolja, itt nem pixelmaszkként szerepel, hiszen a képbájtok *merőlegesen elfordultak*. Ezért itt a pixelek maszkolását mindig a $\$3C4$, $\$02$ regiszterrel oldjuk meg! A bitmaszk regisztert csak azért kell kinullázni, hogy a 4. lépésben a latch átmásolásakor a CPU adat ne módosítsa a másolást.

Az összemontírozáshoz tehát szükségünk van egy BOB-maszkra is, amelyet a *Shape*-ből állítunk elő. Az előállítás logikája egyszerű: ahol a BOB-ban nem háttér színű pixel van, ott a maszk megfelelő bitje 1, ahol pedig háttér színű a pixel, ott a maszk

bitje 0. A maszknak igazodnia kell a BOB tárolásának négybites formátumához, ezért a maszkokat is ebben a formában hozzuk létre. A maszkok előállítását szintén a *Convert_Datas* rutin fogja elvégezni.

Kezdjük el tehát a megvalósítást! A programban 320×240-es felbontást használunk osztott képernyős móddal, az alsó képet 40 pixel magasra állítjuk, a felsőt pedig 200-ra. Első lépésként a videomemóriát felosztjuk a megfelelő részekre, és meghatározzuk a kezdőcímeiket:

Funkció	Normalizált cím	Képetolás cím	Hosszúság
Státuskép	A000:0000h	A000:0000h	3 200 bájt
„A” munkakép	A0C8:0000h	A000:0C80h	16 000 bájt
„B” munkakép	A4B0:0000h	A000:4B00h	16 000 bájt
„C” bufferkép	A898:0000h	A000:8980h	16 000 bájt
BOB adatterület	AC80:0000h	A000:C800h	14 336 bájt

A memóriaterület felosztása után meg kell nézni, hogy milyen változókra és eljárásokra lesz szükség.



BOB256_2.PAS

```
{ $G+, X+}
```

```
Uses Crt, Gr256_2;
```

Definiálunk egy memória és egy memóriamutató típust:

Type

```
MemPtr = ^Memtyp;
```

```
Memtyp = Array[ 0..65534] of byte;
```

A BOB-ok adataihoz is létrehozunk egy típust, amely tárolja egy BOB minden egyes adatát:

```
BOBdata = Record
```

```
  x, y : word; { Koordináták }
```

```
  sha : word; { Mintaszám }
```

```
  adA : word; { "A" kép utolsó cím }
```

```
  adB : word; { "B" kép utolsó cím }
```

```
  Ir : byte; { A BOB mozgásiránya (0..7) }
```

```
  Lp : byte; { Az adott irányban megtett lépések száma }
```

```
end;
```


Szükség van néhány állandóra és inicializált változóra, valamint táblázatokra a mozgathatáshoz és a kirajzoláshoz:

Const

```
BOBsize = Sizeof(BOBdata); { Egy BOB adat hossza }

Bnum = 32; { 32 darab BOB-ot mozgathatunk a képen }

{ Szöveg a státusképre (Az osztott képernyő B része) }
tx : array[ 0..2] of string[ 30] = (
    '256 színű BOB-demó a ',
    'módosított bájt/pixel',
    'szerkezetű képen');
{ Az irányokhoz rendelt elmozdulások }
dx : array[ 0..7] of shortint = (0,1,1,1,0,-1,-1,-1);
dy : array[ 0..7] of shortint = (-1,-1,0,1,1,1,0,-1);

{ A kettősbuffer-technika mutatói }
InacScr : word = $0c80; { Az inaktív kép címe }
InacSEG : word = $a0c8; { Az inaktív kép szegmense }
BuffSEG : word = $a898; { A "C" bufferkép szegmense }
Act : word = $0000; { Az AdA/AdB címmutató }
```

Var

```
{ Általánosan használt változók }
i,j,k : integer;
f : file;
p : MemPtr;

{ BOB adatmező }
BOBs : array [ 1..Bnum] of BOBdata;
{ Normál minta adatmező (ezt kell majd konvertálni) }
Shapes : array [ 0..2047] of byte;
{ BOB maszkok }
Masks : array [ 0..2560] of byte;
{ A BOB-ok kezdőcímének kiszámításához használt táblázat }
cimtab : array [ 0..100] of word;
{ Tároló változók }
r0,r1,r2,r3 : word;
```

Az adatok tehát rendben volnának. Most vegyük sorra az eljárásokat:

Load_Datas: adatbetöltés. Az eljárással a BOB-ok adatait töltjük majd be. A lemez-mellékleten a programhoz a **BOB256_2.DAT** adatfájl tartozik, amely nyolc darab 16×16-os 256 színű BOB-ot tartalmaz. Az adatokat a *Shapes* tömbbe töltjük be.

A *@put_all* ciklus konvertál minden egyes BOB-ot.

```
@put_all:          { A maszk előállítása a BH-ban történik!! }
    mov    ax,0102h    { Kezdés a 01-es bitplane-nel          }
```

A *@put_phases* ciklus előállítja az egy BOB-hoz tartozó négy eltol mintát és ezek maszkjait.

```
@put_phases:
    push    si          { Tároljuk a forráscímet és a kezdő bitmaszkot }
    push    ax
    mov     ch,16        { Egy BOB 16 sorból áll                  }
```

A *@put_bob* egyetlen BOB egyetlen eltolásának konvertálását végzi.

```
@put_bob:
    mov     cl,16        { X irányban pedig 16 (+4) pontból      }
```

A legbelső *@put_line* egyetlen BOB-pixelsor konvertálását végzi el.

```
@put_line:
    out     dx,ax        { A soron következő pont kimaszkolása    }
    cmp     byte ptr [si],0 { Háttér színű a pont ??                }
    jz      @p2          { Ugrás, ha a pont háttér színű          }
    or      bh,ah        { Beírja a bitet a maszkba, ha nem az    }

@p2:
    movsb
    dec     di           { Átmásolás, de a célcím nem nőhet      }
    test    ah,8         { Elértük már a színes bájt szélét ??    }
    jz      @next        { Csak ha elértük a bájt szélét          }
    mov     [di+offset Masks],bh { A maszkadat tárolása        }
    xor     bh,bh        { Nullázza a BH-t az újabb maszk elő-    }
                                állításához
    inc     di           { Most nőhet a cím, mert végigvette a    }
                                négy bitet
    mov     ah,128        { Ez fordul majd be a 0. bitbe (rol)    }

@next:
    rol     ah,1         { A következő plane (újabb pont) ki-    }
                                választása
    dec     cl
    jnz     @put_line    { Kirakja mind a 16 pontot              }

    mov     [di+offset Masks],bh { A maszk kivitele              }
    xor     bh,bh
    inc     di           { A DI növelése 4 pluszpixelt jelent a    }
                                BOB mögött }
```

```

dec    ch
jnz    @put_bob      { Kirakja mind a 16 sort }
pop    ax
pop    si
shl    ah,1          { Eggyel eltolja az egész kirakást
                        (az egész BOB-ot) }
test   ah,16
jz     @put_phases   { Négy fázis van egy BOB-nál }
add    si,256        { Egy BOB 256 bájt hosszú a forrás-
                        adatban }

dec    bl
jnz    @put_all      { Az összes BOB konvertálása }

```

A konvertálás után minden bitplane-t írásra engedélyezünk, és nullázzuk a bitmaszk regisztert.

```

mov    ax,0f02h
out    dx,ax         { Minden bitplane írása engedélyezett }
mov    dx,03ceh
mov    ax,0008h      { A bitmaszk 00000000b (másolást jelent) }
out    dx,ax

pop    DS
End;

```

A *Convert_Datas* meghívása után a *Masks* tömb a *Shapes* tömbben található BOB-ok maszkjaival van feltöltve, és a képmemóriában színes bájtokon tároljuk az összes BOB négy-négy fázisát.

Screen_Init: képernyők inicializálása. Az eljárás az **A** kép tartalmát átmásolja a **B** képre, majd pedig a **B** kép tartalmát a **C** képre.

Procedure Screen_Init; **Assembler;**

```

Asm
cld
mov    ES,SegA000    { Képszegmens }
mov    cx,16000      { Egy kép 16 000 színes bájt }
mov    si,0c80h      { Az A kép kezdőcíme }
mov    di,4b00h      { A B kép kezdőcíme }
rep    seges movsb    { A->B másolás }
mov    cx,16000
rep    seges movsb    { B->C másolás }
End;

```


ShowBOBs: a fő BOB-megjelenítő eljárás. A régi BOB-okat eltünteti az inaktív képről, majd kirakja az új BOB-okat, és átváltja a képernyőt. Ez az eljárás valósítja meg a kettős buffer kezelést, gyakorlatilag az egész program leglényegesebb része a BOB-ok kirajzolása szempontjából.

Procedure ShowBOBs; **Assembler**;

Asm

```

push    DS                { A DS és a BP regisztert tároljuk      }
push    bp

    cld
mov     bp,DS              { A BP az eredeti DS-t fogja adni      }
mov     ES,InacSEG         { Az ES az inaktív képre mutat        }

```

Az *AdA* vagy az *AdB* címek által meghatározott helyeken most a **C** képről vissza kell másolni a képernyő tartalmát az inaktív képre. Ez a BOB-ok eltüntetése az előző helyeiről.

```

    lea    bx,BOBs          { BOB adatok                          }
    add    bx,Act            { Az inaktív kép címei                }
    add    bx,6              { A BX most vagy az AdA-ra, vagy az   }
                             { AdB-re mutat                       }

    mov     dx,03c4h         { Minden plane-t engedélyezünk írásra }
    mov     ax,0f02h
    out     dx,ax

    mov     al,Bnum          { Az AL számolja a BOB-okat          }
@restore_all:
    mov     DS,bp            { Az eredeti DS-t veszi,itt vannak az }
                             { adatok                              }
    mov     di,[bx]          { Beolvassuk a képcímet (AdA vagy AdB) }
    mov     si,di            { A képcímek offset részei azonosak   }
    mov     DS,BuffSEG       { A DS most a C képre mutat          }

    { A DS:SI most a bufferképre mutat (C)                          }
    { Az ES:DI pedig az inaktív képre (A vagy B)                    }

    mov     cx,16            { 16 sor átmásolása következik;      }
                             { ez egy BOB eltüntetése              }

@restore_bytes:
    movsb                    { 5*4 pixel, azaz 5 színes bájt másolása }
    movsb
    movsb
    movsb
    movsb

```

```

add    di,75          { Növeljük a címeket }
add    si,75
loop   @restore_bytes { Eltüntetünk egy BOB-ot }
add    bx,BOBsize     { Az adatmutatót a következő adatra
                        állítjuk }
dec    al             { Csökkentjük a BOB-számlálót }
jnz    @restore_all   { Vissza, ha van még eltüntetendő BOB }

```

A `@restore_all` ciklus lefutása után az inaktív kép helyreállt. Most a BOB adatai alapján ki kell számítani a képcímet, valamint a BOB-shape és a maszk kezdőcímét.

```

mov     DS,bp          { Az eredeti adatszegmens }

mov     cx,Bnum        { A CX számolja a BOB-okat }
lea     bx,BOBs        { A BX mutat a BOB adataira (BOBs tömb) }
@put_all_BOB:         { Ez a ciklus végzi az összes BOB
                        kirajzolását }

{ Először a képcímet számítjuk ki az X,Y rekordmezők alapján }

mov     si,Act         { A címkorrigáló érték (0/2) az SI-be }
mov     ax,80          { Képcímszámítás (a kép logikai
                        hossza 80) }

mov     di,[bx+2]      { Y }
mul     di             { Y * Lsize }
mov     di,[bx]        { X }
shr     di,2
add     di,ax          { A DI-ben előáll a képcím }
mov     [bx+si+6],di   { AdA vagy AdB feltöltése a képcímmel }

{ Most meghatározzuk a Shape és a maszk címét az Sha mező alapján }

lea     si,cimtab      { Mintacím számítás }
mov     ah,[bx]        { AH = lo(X) }
and     ah,3           { Az X alsó két bitje adja a bitpozíciót
                        (AH) }
shl     ah,1           { AH = 0,2,4,6 (az X biteltolás két-
                        szerese) }

mov     al,[bx+4]      { Mintaszám (Sha) }
shl     al,3           { Mintaszám*4*2 }
or      al,ah          { A bitek összemácsolása }
xor     ah,ah
add     si,ax          { Az SI a minta eltolására mutat }
mov     si,[si]        { SI a minta eltolását adja }

```



```

mov    r0,bx                { Tároljuk az adatmutatót (BX) és a
                             BOB-számlálót (CX) }
mov    r1,cx

lea    bx,Masks             { Meghatározzuk a BOB maszkjának kezdő-
                             címét }
add    bx,si                { A BX most a maszkadatra mutat
                             (DS:[bx]->Maszk) }
add    si,0c800h            { Az SI a BOB adatra mutat }
sub    si,InacScr           { Korrigáljuk a címet, hogy az A vagy a
                             B képre mutasson }
                             { Az SI-ben a cím most az inaktív képre
                             mutat (ES:[SI]) }

mov    al,2
mov    dx,03c4h            { A Write Plane Enable regisztert hasz-
                             náljuk,
                             { ezzel maszkoljuk a biteket (plane-eket)}

```

A címek kiszámítása után a BOB-ot ki kell rakni az inaktív képre. A *@putBOB* ciklus egyetlen BOB kirajzolását végzi el. A kirajzolás a vízszintes sorok egymás utáni kimásolásával történik, az eljárás a színes bájtokat a maszk alapján montírozza össze. Mivel egy eltolt BOB 20 pixel széles, ezért 5 darab színes bájtot kell átmásolni egy sorban. A maszk formája megegyezik a Shape formájával (négybites adatok, 5 bájt/sor).

```

mov    cx,16                { 16 sor kirajzolása következik }
@putBOB:
mov    ah,[bx]              { Beolvassa a maszk adatot, majd }
out    dx,ax                { kiviszi a Write Plane Enable regiszterbe }
seges  movsb                { Kirakja a BOB pixeleit (első képbájt) }
mov    ah,[bx+1]
out    dx,ax
seges  movsb                { Második képbájt }
mov    ah,[bx+2]
out    dx,ax
seges  movsb                { Harmadik képbájt }
mov    ah,[bx+3]
out    dx,ax
seges  movsb                { Negyedik képbájt }
mov    ah,[bx+4]
out    dx,ax
seges  movsb                { Ötödik képbájt }
add    bx,5                 { Növeljük a maszkmutatót }
add    di,75                { Növeljük a képcímet }
loop   @putBOB              { Kirajzoljuk a BOB 16 sorát }

```

```

mov    bx,r0          { Visszahívjuk az adatmutatót és a BOB-
                        számlálót }
mov    cx,r1
add    bx,BOBsize     { Az adatmutató a következő BOB rekordjára
                        áll }
dec    cx
jnz    @put_all_BOB   { Vissza, minden BOB kirajzolása }

```

Az összes BOB kirajzolása után az új kép elkészült az inaktív képernyőn. Most már csak át kell váltani a két képet, és ki kell várni egy vertikális visszafutást, hogy a másik kép biztosan eltűnjön.

```

mov    al,0ch
mov    dx,03d4h
mov    bx,InacSCR      { A kép címét az InasSCR változó tárolja }
mov    ah,bh           { A felső bájt }
out    dx,ax           { A felső bájt a $3D4, $0C regiszterbe }
mov    ah,bl           { Az alsó bájt }
inc    al
out    dx,ax           { Az alsó bájt a $3D4, $0D regiszterbe
                        { Mutatók és változók cseréje } }

xor    Act,2           { Az AdA/AdB korrekció váltása (0/2) }
xor    InacSCR,4780h   { Az inaktív kép címének váltása }
xor    InacSEG,0478h   { Az inaktív kép szegmensének váltása }

{ Várakozás egy vertikális visszafutásra }

mov    dx,$03da
@Wait:
in     al,dx
and    al,8
jz     @Wait          { Várakozás az elektronsugár-visszafutásra }

pop    bp             { Visszaírjuk a DS és a BP eredeti értékeit }
pop    DS
End;                { Kilépés a megjelenítőből }

```

Hát ennyi lenne a BOB-ok megjelenítése. Elviekben ez a rutin semmiben sem különbözik a 16 színű grafikánál megismert kettősbuffer-technikától, csak a képszerkezet miatt módosult némileg a BOB-ok kirajzolása. Ha visszalapozunk az egyszínű és a tizenhat színű kettősbuffer-módszer megvalósításához, akkor látható, hogy ez a megjelenítő mindkettőnél gyorsabb, mert kevesebb műveletet kell elvégezni a BOB adatokkal. Az egyszínű megjelenítő esetében a BOB adatok forgatása vesz el sok időt, míg a többszínűnél a BOB-ok bitplane-enkénti kirajzolása az időigényes művelet. Nem véletlen tehát, hogy ez a képszerkezetet a leggyakoribb a játékprogramokban, hiszen a sok szín mellett még a BOB-ok kezelése is viszonylag gyors.

InstallSplit: az osztott képernyő bekapcsolása. Az eljárás a *Line Compare* CRTC regisztert állítja be \$018f-re, azaz 399-re. (Mivel a rastersorok számozása nullával kezdődik, ezért ez valójában a 400-adik rastersort jelenti, persze csak a mi megszo-
kott, 1-gyel kezdődő sorszámozásunkban. Említettük már, hogy a számítástechniká-
ban általában nullával kezdődik a sorozatok számozása, ami egyébként logikus is.)

Procedure InstallSplit; **Assembler**;

Asm

```

mov    dx,03d4h
mov    ax,8f18h           { A Line Compare alsó nyolc bitje      }
out    dx,ax
mov    ax,8009h           { A Line Compare 9. bitje nulla (6.bit) }
out    dx,ax
mov    ax,0c0ch           { A kép kezdőcímét az A képre állítjuk }
out    dx,ax
mov    ax,800dh
out    dx,ax

```

End;

MoveBOBs: a BOB-ok mozgatása a képernyőn. A mozgatási algoritmus a már meg-
ismert nyolcírányú véletlenszerű mozgatást valósítja meg. (Emlékeztetőül: véletlen-
szerűen generálunk egy irányt (0..7) és hozzá valahány lépést. A BOB mindaddig
errefelé mozog, amíg a lépés elfogy, vagy a BOB falnak ütközik.)

Procedure MoveBOBs;

Begin

```

for i:=1 to Bnum do      { Minden BOB-ot mozgatunk              }

```

```

  with BOBs[ i] do

```

```

    Begin

```

```

      dec(lp);           { Csökkentjük a lépésszámot            }

```

```

      if lp=255 then

```

```

        Begin           { Ha a lépésszám lejárt, akkor új irány
                           és lépés                                }

```

```

          ir:=random(8); { Egy véletlenszerű irány              }

```

```

          lp:=10+random(30); { Egy lépésszám 10 és 39 között          }

```

```

        End;

```

```

      { A képszélek ellenőrzése }

```

```

      if x<5 then ir:=2;

```

```

      if x>300 then ir:=6;

```

```

      if y<5 then ir:=4;

```

```

      if y>180 then ir:=0;

```

```

      inc(x,dx[ ir]);

```

```

      inc(y,dy[ ir]);

```

```

    End;

```

End;

Végül nézzük a főprogramot! Itt megteesszük az összes előkészületet a kettősbuffer-módszerhez, majd elkezdjük mozgatni a BOB-okat.

Inicializáljuk a BOB-shape címszámításához használt táblázatot. Minden egyes BOB 80 színes bájtot foglal el a képmemóriában, ezért a BOB-ok egymástól 80 bájt távolságra vannak. Mivel a BOB-okat eltolva tároljuk, ezért egy teljes BOB-ot 4*80 bájt tárol.

BEGIN

```
for i:=0 to 100 do cimtab[ i ]:=80*i;
```

Betöltjük a Shape adatokat és a háttérképet, ez utóbbi egy Mandelbrot-fraktál részlet.

```
Load_Datas;           { A Shape adatok betöltése }
getmem(p,64000);      { Helyfoglalás a háttérkép számára
                      (320x200/256) }
Load_Picture;         { Betöltjük a háttérképet }
```

Most már be lehet kapcsolni a *módosított bájt/pixel* képformátumot, hogy konvertálni tudjuk a Shape adatokat, és meg tudjuk jeleníteni a háttérképet.

```
_320x240;

for i:=0 to 199 do      { A háttérképet pixelenként kell kirakni
                        a képernyőre }

  for j:=0 to 319 do
    putpixel(j,i+40,p^[ i*320+j ] );
freemem(p,64000); { Felszabadítjuk a lefoglalt területet }

{ Inicializáljuk a BOB-ok adatmezőit }
for i:=1 to Bnum do
  with BOBs[ i ] do
    Begin
      x:=10+random(290); { Egy véletlenszerű X koordináta }
      y:=10+random(170); { Egy véletlenszerű Y koordináta }
      sha:=i and 7;      { A minta száma }
      adA:=0;            { A képcímek nullázása }
      adB:=0;
      ir:=random(8);     { Kezdőirány }
      lp:=random(30)+10; { Kezdőlépés }
    End;
```

Az osztott képernyő alsó részére a *GR256_2* egység eljárásaival tudunk írni és rajzolni. Kirajzoljuk az alsó státusképet:


```

Bbar(0,0,319,39,0);
Cbar(4,4,315,35,1);
Ink:=14;
OutStrXY(73, 6,tx[ 0] );
Ink:=0;
OutStrXY(74, 6,tx[ 0] );
Ink:=15;
OutStrXY(70,16 ,tx[ 1] );
Ink:=4;
OutStrXY(71,16 ,tx[ 1] );
Ink:=0;
OutStrXY(90,26,tx[ 2] );
Ink:=14;
OutStrXY(91,26,tx[ 2] );
for i:=1 to 33 do
  begin
    ink:=i+32;
    line(280+i,5,251+i,34);
    line(34+i,5,5+i,34);
  end;

```

Konvertáljuk a *Shape* adatokat, a képmemóriában létrehozzuk a színes BOB-okat, és a BOB-ok maszkjaival feltöltjük a *Masks* tömböt.

```

Convert_Datas(Shapes,$ac80,8);

Screen_Init;      { Inicializáljuk a B és a C képet }
InstallSplit;     { Bekapcsoljuk az osztott képernyős módot }

Showbobs;         { Egy álló kép megjelenítése (a BOB-ok mozdulat-
                  lanok) }
readkey;          { Várakozás egy billentyűre }

```

A BOB-okat mostantól mozgatjuk a képen. A program egy billentyű lenyomására kilép a mozgatásból.

```

repeat
  MoveBOBs;      { Elmozdítás }
  ShowBOBs;      { Megjelenítés }
until keypressed;
while keypressed do readkey;
{ Kilépés, visszatérés text módba }
textmode(c80);
END.

```

A program egyetlen hátránya, hogy a szabad 14 336 bájt képmemóriában kevés színes *Shape* adatot tudunk tárolni. Azokon a VGA kártyákon, ahol legalább 512K memória van, ezt a problémát ki lehet küszöbölni, ha a videoterület nagyságát 128K-ra állítjuk a *\$3CE*, *\$06 Memory Mode* regiszterrel. Ekkor a B000:0000 címtől kezdve is 64K színes bájtterületet használhatunk, amelyet megjeleníteni ugyan nem lehet, de mind a C bufferkép, mind a Shape adatok tárolására alkalmas. Ebben az esetben módosítani kell a minták kezdőcímének kiszámítását a *ShowBOBs* rutinban. Hogy ezt melyik VGA kártya tudja megvalósítani, azt csak próbával lehet eldönteni.

9.12. Teljes képernyő görgetése

A 256 színű képernyőről szóló fejezet utolsó témájához érkeztünk. A karakteres képnél már megismertük a globális és az ablakkép fogalmát, és példát is láttunk az alkalmazásukra. A tizenhat színű bitplane szervezésű képek esetében is alkalmaztuk a módszert. A *normál bájt/pixel* képszerkezetnél viszont nem használhattuk, mert túl nagy volt a kép címtartománya. A *módosított bájt/pixel* a *normál bájt/pixel* formátummal ellentétben alkalmas a teljes képernyős görgetésre. A most következő példában egy 640×400/256-os globális képet hozunk létre, míg az ablakkép felbontása 320×200 lesz, azaz az ablakkép a globális kép egy negyedét mutatja. Az ablakkép pozicionálását a *\$3D4*, *\$0C*, *\$0D képkezdőcím* regiszterrel és a *\$3C0*, *\$13 horizontális eltolás* regiszterrel tudjuk megvalósítani. Ha visszalapozunk a regiszterek leírásához, akkor láthatjuk, hogy a horizontális eltolást a többi módtól eltérően kell értelmeznünk a 256 színű módban:

A <i>\$3C0</i> , <i>\$13</i> értéke	Pixeles eltolás a 256 színű módban
0	0
1	–
2	1
3	–
4	2
5	–
6	3
7	–

A regiszter tehát eleve figyelembe veszi, hogy a képbájtokon itt négy pixelt tárolunk. A képcím és az eltolás kiszámítása a következő két képlettel lehetséges:

$$\text{képcím} = 160 * Y + (X \text{ SHR } 2)$$

$$X \text{ eltolás} = (X \text{ AND } 3) \text{ SHL } 1$$

A *ShowPic* eljárásnak tehát ezt a két számítást kell elvégeznie. A programban egy Mandelbrot-fraktál részletet jelenítünk meg a globális képen, és ezen tudjuk pozicionálni a látható képernyőt. A látványosság növelése érdekében a paletta fejezetben megismert színrotációt is alkalmazzuk, így meglehetősen érdekes és szép ábrát kapunk. A lemezmellékleten a **vgascrn.003** fájl tartalmazza a fraktál képét, mégpedig az első 768 bájt a palettát, a következő 256 000 bájt pedig a 640×400-as kép pixeleit bájtonként.



SCROLL.PAS

```
{ $g+,x+}
```

```
Uses Gr256_2,Crt,Crtplus;
```

Változók és állandók deklarálása:

Const

```
palrot_speed = 5;    { A palettarotáció sebessége (1=leggyorsabb) }
```

```
{ Pixelpozíciók egy képbájton belül az Mpx számára }
```

```
PixelPos : Array [ 0..3] of byte = ($11,$22,$44,$88);
```

```
{ Az ablakkép koordinátái }
```

```
X : word = 0;
```

```
Y : word = 0;
```

```
palx : byte = palrot_speed; { Palettarotáció számláló }
```

Var

```
{ Általánosan használt változók }
```

```
i,j      : integer;
```

```
f        : file;
```

```
Pal      : Array[ 0..767] of byte;    { A paletta adatai }
```

```
Sline    : Array[ 0..639] of byte;    { Egy pixelsor adatai }
```

```
RGBr     : Array[ 0..2] of byte;      { Egy RGB adat tárolása }
```

Eljárások:

Palanim: a paletta rotációja. Ez az eljárás végzi az RGB regiszterek módosítását. A képhez tartozó paletta 254 színű, amelyet a *Pal* tömb tartalmaz. A tömb első 254*3 elemét kell mozgatni, hogy minden szín egy regiszterrel lejjebb kerüljön majd. Az elmozdítás után az értékeket kiírjuk az RGB regiszterekbe, de a háttér színét (0. RGB regiszter) nem módosítjuk. Az eljárást a *ShowPic* ablakkép-pozicionáló eljárás hívja

meg mindig éppen egy vertikális visszafutás után, ezért a paletta átírása nem okoz zavart a képernyőn (legalábbis a gyors gépeken). A rotáció sebességét — tehát a *Palanim* rutin hívását — a *palrot_speed* állandóval tudjuk beállítani.

Procedure Palanim;

Begin

```

move(Pal,RGBr,3);           { Az első RGB adat az RGBr változóba
                             kerül }
move(Pal[ 3] ,Pal,759);     { A paletta 254 eleme lejjebb mozog }
move(RGBr,Pal[ 759] ,3);    { Az RGBr adat a 254. RGB adatba kerül}
asm                        { Az RGB regiszterek beállítása követ-
                             kezik }

    cld
    mov  dx,03c8h           { Az RGB regiszterek írása }
    mov  al,1               { A háttér nem módosulhat }
    out  dx,al
    lea  si,Pal
    mov  cx,253*3           { 253 regisztert kell átírni }
    inc  dx
    rep  outsb              { Az adatok kivitele az RGB regisz-
                             terekbe }

```

end;

End;

Mpx: módosított pixelrajzolás. Az eljárás a globális kép *X,Y* helyére kirakja a *Color* színű pontot. Az eljárás működése megegyezik a *Gr256_2.PutPixel* működésével, csak a kép nagysága miatt az *Y* koordinátát itt 160-nal kell szorozni, mivel a globális képnek ez a logikai hossza. Az *Mpx*-et a fraktálkép kirajzolására használjuk.

Procedure Mpx(*X,Y,Color:word*); **Assembler;**

Asm

```

mov  ax,160                { A kép logikai hossza most 160 (640 pixel) }
mul  Y
mov  di,X
shr  di,2                  { Az X alsó két bitje nem kell }
add  di,ax
mov  ES,SegA000            { ES:DI mutat a képbájtra }
lea  bx,PixelPos
mov  al,byte [ X]
and  al,3                  { Az X alsó két bitje a bitpozíciót adja }
xlat
mov  dx,03c4h              { Write Plane Enable regiszter }
mov  ah,al
mov  al,2
out  dx,ax                 { Kivisszük a bitmaszket }
mov  ax,Color
mov  ES:[ di] ,al          { Kirajzoljuk a pontot }

```

End;

ShowPic: az ablakkép pozicionálása a globális képen. Az eljárás az *X,Y* koordináták alapján a fentebb ismertetett képletekkel kiszámolja a képcímet és az *X* pixeles eltolást, majd módosítja a *\$3D4*, *\$0C*, *\$0D* képkézdőcím és a *\$3C0*, *\$13* horizontális eltolás regisztereket. A regiszterek módosítása után csökkenti a *palx* változót, amelynek a nulla értéke jelzi, hogy a palettát rotálni kell. Az eljárás ekkor elvégzi a paletta rotációját a *Palanim* eljárás meghívásával.

Procedure ShowPIC (*X,Y:word*); **Assembler;**

Asm

```

mov  bx,X           { Az X koordináta a BX-be }
mov  cl,bl          { Az X alsó két bitjét a CL-ben tároljuk el }
shr  bx,2           { Képcímszámítás }
mov  ax,160
mul  Y
add  bx,ax          { A BX-ben megjelenik a képcím }
{ A regiszterek módosítása }
mov  dx,03d4h
mov  al,0ch
mov  ah,bh
out  dx,ax          { $3D4, $0C: képcím felső bájt }
inc  al
mov  ah,bl
out  dx,ax          { $3D4, $0D: képcím alsó bájt }

and  cl,3           { Ax X pixeles eltolás a CL-be kerül (alsó két bit) }
shl  cl,1           { CL = 0,2,4,6 }
{ Várakozni kell egy vertikális visszafutásra }
mov  dx,03dah       { Várakozás a vertikális visszafutásra }
@wait:
in   al,dx
and  al,8
jz   @wait
mov  dl,0c0h
mov  al,13h         { Horizontális eltolás regiszter $3C0,$13 }
out  dx,al
mov  al,cl          { Az eltolás kivitele a regiszterbe }
out  dx,al
mov  al,020h        { A kép engedélyezése }
out  dx,al
dec  palx           { A palettarotáció-számláló csökkentése }
jnz  @exit          { Kigrás, ha palx<>0 }
mov  palx,palrot_speed { A palx változó újratöltése }
Call Palanim        { A paletta rotálása }
@exit:
End;
```

A főprogramban be kell állítanunk a globális képet, és a lemezről be kell töltenünk a fraktál képét.

```

BEGIN
  _320x200; { A 320x200/256 felbontást használjuk }
  portw[ $3d4] := $5013; { A kép logikai hosszát 160 bájttra
                          állítjuk }
  { A globális kép vízszintesen most már 160 bájt (640 pixel) }

  { Beolvassuk a palettát és a képet }
  assign(f, 'vgascrn.003'); { A 640x400/256-os kép neve scrn.001" }
  reset(f,1);
  blockread(f, Pal, 768); { A fájl első 768 bájtja a paletta }
  asm
    mov ax, 1012h { A paletta beállítása a BIOS rutinnal}
    xor bx, bx
    mov cx, 256 { 256 színt állítunk be }
    push DS
    pop ES
    lea dx, Pal { ES:DX adja a palettaadatok címét }
    int 10h { A BIOS palettabeállító meghívása }
  end;

```

A paletta beolvasása és beállítása után a lemezen található képet soronként beolvasuk az *Sline* tömbbe, és kirajzoljuk a képernyőre. Az *Sline* tömböt azért használjuk átmeneti tárolóként, mert a *blockread* eljárás jóval gyorsabb, mint a *read*.

```

for i:=0 to 399 do { A kép 400 soros (függőleges méret) }
  Begin
    blockread(f, Sline, 640); { Egy vízszintes sor beolvasása }
    for j:=0 to 639 do Mpx(j, 399-i, Sline[j]);
                                { A pixelek kirajzolása }
  End;

  Close(f); { Lezárjuk a fájlt }

```

A beolvasás után a képek inicializálása elkészült, és a képernyőn látható a globális kép bal felső negyede. Most a **Crtplus** egységben található módosított billentyűzetmegszakítást állítjuk be:

```

InstallKeyIRQ; { A saját billentyűzetmegszakítás aktiválása }

{ A kép most a kurzorbillentyűvel mozgatható }
repeat
  if Ktb[ 72] and (y>0) then dec(y, 2);
  if Ktb[ 77] and (x<320) then inc(x, 2);

```



```
if Ktb[ 75] and (x>0) then dec(x,2);  
if Ktb[ 80] and (y<199) then inc(y,2);  
Showpic(x,y);  
Delay(10);  
until Ktb[ 1];      { Az ESC-re kilép a ciklusból }
```

A képet tetszőlegesen lehet tologatni a kurzorbillentyűkkel, a paletta pedig folyamatosan változik, a színek cserélődnek. Az ESC lenyomására az eredeti billentyűzetrutint állítjuk vissza, text módot állítunk be, és kilépünk.

```
UninstallKeyIRQ; { Vissza az eredeti billentyűzetmegszakítást }  
{ Text mód, kilépés }  
textmode(c80);  
END.
```

10. Nem szabványos videomódok



NONSTD

Ebben a fejezetben arról lesz szó — röviden —, hogy mi mindent tud a VGA kártya, amit a gyártók nem közölnek, és amit a BIOS sem támogat. Az előző fejezetekben sokszor előfordult már, hogy nem szabványos felbontásokat használtunk. Most meg-
nézzük az összes ilyen esetet és a beállításuk módját is. A nem szabványos videomó-
dok beállításakor a legnagyobb szerepe a CRTC regisztereknek van, hiszen a képer-
nyő méretét és szerkezetét jórészt ezek határozzák meg. Szerencsére az összes regisz-
ter beállítását le tudjuk kérdezni (az EGA kártyákkal ellentétben), így ismertté válnak
a szabványos módokhoz rendelt beállítások. A lekérdezés után a rendelkezésünkre áll
majd egy olyan adattáblázat, amelynek elemeit kombinálva sok-sok variációt elő tu-
dunk állítani. A lekérdezést a lemezmellékleten található **VGAREGS.PAS** program-
mal végezhetjük el, amely létrehoz egy **VGA_REG.TXT** szövegfájlt, amelyben táb-
lázatos formában jeleníti meg az eredményeket.

A program futtatásakor a következő táblázat jelenik meg:

VGA	320x200	640x200	640x350	640x480	320x200	80x25tx	40x25tx
regs.	16 col	16 col	16 col	16 col	256 col	16 col	16 col
\$3d4: CRTC regiszterek:							
00:	2d(45)	5f(95)	5f(95)	5f(95)	5f(95)	5f(95)	2d(45)
01:	27(39)	4f(79)	4f(79)	4f(79)	4f(79)	4f(79)	27(39)
02:	28(40)	50(80)	50(80)	50(80)	50(80)	50(80)	28(40)
03:	90(144)	82(130)	82(130)	82(130)	82(130)	82(130)	90(144)
04:	2b(43)	54(84)	54(84)	54(84)	54(84)	55(85)	2b(43)
05:	80(128)	80(128)	80(128)	80(128)	80(128)	81(129)	a0(160)
06:	bf(191)	bf(191)	bf(191)	0b(11)	bf(191)	bf(191)	bf(191)
07:	1f(31)	1f(31)	1f(31)	3e(62)	1f(31)	1f(31)	1f(31)
08:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
09:	c0(192)	c0(192)	40(64)	40(64)	41(65)	4f(79)	4f(79)
0a:	00(0)	00(0)	00(0)	00(0)	00(0)	0d(13)	0d(13)
0b:	00(0)	00(0)	00(0)	00(0)	00(0)	0e(14)	0e(14)
0c:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
0d:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
0e:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
0f:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
10:	9c(156)	9c(156)	83(131)	ea(234)	9c(156)	9c(156)	9c(156)
11:	8e(142)	8e(142)	85(133)	8c(140)	8e(142)	8e(142)	8e(142)

VGA	320x200	640x200	640x350	640x480	320x200	80x25tx	40x25tx
regs.	16 col	16 col	16 col	16 col	256 col	16 col	16 col

\$3d4: CRTC regiszterek:

12:	8f(143)	8f(143)	5d(93)	df(223)	8f(143)	8f(143)	8f(143)
13:	14(20)	28(40)	28(40)	28(40)	28(40)	28(40)	14(20)
14:	00(0)	00(0)	0f(15)	00(0)	40(64)	1f(31)	1f(31)
15:	96(150)	96(150)	63(99)	e7(231)	96(150)	96(150)	96(150)
16:	b9(185)	b9(185)	ba(186)	04(4)	b9(185)	b9(185)	b9(185)
17:	e3(227)	e3(227)	e3(227)	e3(227)	a3(163)	a3(163)	a3(163)
18:	ff(255)	ff(255)	ff(255)	ff(255)	ff(255)	ff(255)	ff(255)

VGA	320x200	640x200	640x350	640x480	320x200	80x25tx	40x25tx
regs.	16 col	16 col	16 col	16 col	256 col	16 col	16 col

\$3cc: Kimeneti állapot regiszter (\$3c2):

63(99)	63(99)	a3(163)	e3(227)	63(99)	67(103)	67(103)
---------	---------	---------	---------	---------	---------	---------

\$3c4: Sequencer regiszterek:

00:	03(3)	03(3)	03(3)	03(3)	03(3)	03(3)	03(3)
01:	09(9)	01(1)	01(1)	01(1)	01(1)	00(0)	08(8)
02:	0f(15)	0f(15)	0f(15)	0f(15)	0f(15)	03(3)	03(3)
03:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
04:	06(6)	06(6)	06(6)	06(6)	0e(14)	02(2)	02(2)

\$3ce: Grafikai vezérlőregiszterek:

00:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
01:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
02:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
03:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
04:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
05:	00(0)	00(0)	00(0)	00(0)	40(64)	10(16)	10(16)
06:	05(5)	05(5)	05(5)	05(5)	05(5)	0e(14)	0e(14)
07:	0f(15)	0f(15)	0f(15)	0f(15)	0f(15)	00(0)	00(0)
08:	ff(255)	ff(255)	ff(255)	ff(255)	ff(255)	ff(255)	ff(255)

\$3ca: Vertikális szinkron olvasás regiszter:

00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
--------	--------	--------	--------	--------	--------	--------

\$3da: Státuszregiszter:

30(48)	30(48)	30(48)	31(49)	30(48)	30(48)	31(49)
---------	---------	---------	---------	---------	---------	---------

\$3c0: Attribute regiszterek:

00:	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)	00(0)
01:	01(1)	01(1)	01(1)	01(1)	01(1)	01(1)	01(1)
02:	02(2)	02(2)	02(2)	02(2)	02(2)	02(2)	02(2)
03:	03(3)	03(3)	03(3)	03(3)	03(3)	03(3)	03(3)
04:	04(4)	04(4)	04(4)	04(4)	04(4)	04(4)	04(4)
05:	05(5)	05(5)	05(5)	05(5)	05(5)	05(5)	05(5)
06:	06(6)	06(6)	14(20)	14(20)	06(6)	14(20)	14(20)
07:	07(7)	07(7)	07(7)	07(7)	07(7)	07(7)	07(7)
08:	10(16)	10(16)	38(56)	38(56)	08(8)	38(56)	38(56)
09:	11(17)	11(17)	39(57)	39(57)	09(9)	39(57)	39(57)

VGA	320x200	640x200	640x350	640x480	320x200	80x25tx	40x25tx
regs.	16 col	16 col	16 col	16 col	256 col	16 col	16 col

\$3c0: Attribute regiszterek:

0a:	12 (18)	12 (18)	3a (58)	3a (58)	0a (10)	3a (58)	3a (58)
0b:	13 (19)	13 (19)	3b (59)	3b (59)	0b (11)	3b (59)	3b (59)
0c:	14 (20)	14 (20)	3c (60)	3c (60)	0c (12)	3c (60)	3c (60)
0d:	15 (21)	15 (21)	3d (61)	3d (61)	0d (13)	3d (61)	3d (61)
0e:	16 (22)	16 (22)	3e (62)	3e (62)	0e (14)	3e (62)	3e (62)
0f:	17 (23)	17 (23)	3f (63)	3f (63)	0f (15)	3f (63)	3f (63)
10:	01 (1)	01 (1)	01 (1)	01 (1)	41 (65)	0c (12)	0c (12)
11:	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)
12:	0f (15)	0f (15)	0f (15)	0f (15)	0f (15)	0f (15)	0f (15)
13:	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)	08 (8)	08 (8)
14:	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)	00 (0)

Ez a táblázat így, önmagában még nem sokat mond, de ennek alapján már kombinálhatjuk a különböző beállításokat, létrehozhatjuk a nem szabványos videomódokat.

A lemezmellékleten található programok segítségével az alábbi módokat lehet beállítani:

Textmódok	Grafikus 16 színű	Grafikus 256 színű
<i>Szabványos:</i>		
40x25 TX	320x200 BPL	320x200 B/P
80x25 TX	640x200 BPL	
	640x350 BPL	
	640x480 BPL	
<i>Nem szabványos:</i>		
80x28 TX	320x240 BPL	320x200 M B/P
80x50 TX	320x350 BPL	320x240 M B/P
80x30 TX	320x400 BPL	320x350 M B/P
80x34 TX	320x480 BPL	320x400 M B/P
80x60 TX	640x240 BPL	320x480 M B/P
90x25 TX	640x400 BPL	360x200 M B/P
90x28 TX	720x400 BPL	360x240 M B/P
90x30 TX	720x480 BPL	360x350 M B/P
90x34 TX		360x400 M B/P
		360x480 M B/P

A táblázatban a következő jelöléseket használtuk:

- TX szöveges mód (word címezsmód)
- BPL bitplane szervezésű képszerkezet
- B/P normál bájt/pixel szervezésű képszerkezet
- MB/P módosított bájt/pixel szervezésű képszerkezet

A táblázat jól mutatja, hogy a BIOS készítői milyen sok mindent kihagytak a meglévő lehetőségekből. Azt is el kell mondani, hogy még ez a táblázat sem teljes, hiszen ezekben az esetekben még el sem térünk a szabványos regiszterbeállítási értékektől, csupán kombináljuk a különböző lehetőségeket. Ha viszont még a szabványtól eltérő értékekkel is dolgoznánk, akkor lehetőség nyílna ennél több videomód beállítására is!

Figyelem!!!

Mivel a továbbiakban közölt módok egyike sem szabványos, ezért előfordulhat, hogy néhány régebbi típusú (főként 8-bites) VGA kártyán egyes beállítások sikertelenek lesznek! A beállítások előtt ezért érdemes minden egyes kártyán lefuttatni a **VGAREGS.PAS** programot, hogy megvizsgálhassuk a regisztereknek a fenti táblázattól való esetleges eltéréseit, és megfelelően korrigálni tudjuk a beállításokat.

10.1. Szöveges módok

A szöveges módok esetében igazából nem is lehet meghatározni, hogy hányféle felbontást lehetne beállítani, hiszen tudjuk, hogy a VGA kártya 1 és 32 között bármekkora karaktermagasságot megenged, így azután a betűkészlet magasságától és a függőleges felbontástól függően nagyon sok felbontást el tudunk érni. A szöveges módok programozásánál már megismertük ezeket a rutinokat, így itt most csak az elvi megoldásokat nézzük meg, és azt, hogy mire kell ügyelni. Az átváltás során két dolgot kell tennünk:

1. Meg kell változtatnunk a függőleges felbontást.
2. Meg kell változtatnunk a karakterkészlet magasságát.

A függőleges felbontás 200, 240, 350, 400 és 480 lehet, a karakterek magassága pedig 1 és 32 között tetszőlegesen változhat. Ha feltételezzük azt, hogy a karakterek 7-pixeles magasságtól felfelé már olvashatóak, akkor összesen — szabványos értékeket használva — $5 \cdot 26 = 130$ -féle függőleges felbontást tudunk megadni, 6.25-től ($200/32$) egészen 68.57-ig ($480/7$). Ha ehhez még hozzávesszük, hogy vízszintesen 40 és 80 karakter (45 és 90 karakter) állítható be, akkor a lehetséges karakteres módok száma: $4 \cdot 130 = 520$!

Példaképpen nézzük meg a 80×30-as mód beállítását. A program megegyezik a karakteres képről szóló fejezetben ismertetettel.

A 80x30-as mód beállítása:

Procedure Char80x30;**Assembler;**

Asm

```

mov ax,0003h
int 10h           { A normál 80x25-ös mód beállítása }
mov dx,03d4h      { $03D4 = elektronsugár regiszterek }
mov ax,0e11h      { Engedélyezzük a regiszterek írását }
out dx,ax         { $11-es regiszter: a 7. bit 0-ba ! }
mov cx,18h        { $18 = 24 db regisztert kell átírni }
xor al,al
lea si,@dat480    { A tárolt adatok címe a memóriában }
@cikl:
mov ah,CS:[si]    { A következő adat beolvasása }
out dx,ax         { A regiszter átírása }
inc si
inc al
loop @cikl        { Átírja mind a 24 regisztert }
jmp @vege         { Visszatérés }
@dat480:
db 5fh,4fh,50h,82h,55h,81h
db 0bh,3eh,00h,4fh,0dh,0eh,00h,00h,00h,00h
db 0eah,8ch,0dfh,28h,1fh,0e7h,04h,0a3h,0ffh
@vege:
End;

```

Az eljárásban szereplő *@dat480* táblázat a program lényege. A táblázatban összekombináltuk a 80 karakteres vízszintes felbontáshoz tartozó horizontális regiszterek és a 480-as függőleges felbontáshoz tartozó vertikális regiszterek értékét. Először a BIOS funkcióval beállítjuk a 80x25-ös módot, majd a CRTC regiszterek átírása következik. Ehhez elsőként a *\$3D4*, *\$11 Vertical Retrace End* regiszter 7. bitjét kell törölni, ugyanis ez engedélyezi a 0..7 regiszterek írását. Ezt követően a *@cikl* ciklus sorban beállítja a regisztereket. Látható, hogy ebben az esetben változatlanul hagytuk a karakterek magasságát, tehát a fejezet elején említett két feladatból most csak egyet hajtottunk végre.

A felbontás növelésével általában az a cél, hogy több karaktert tudjunk megjeleníteni a képen, de a szöveg ettől még olvasható maradjon. Ha például beállítjuk a 80x60-as módot, akkor a betűk már túl kicsik, és az ilyen szöveg olvasása megerőltető. Ezért most nézzük meg a vízszintes méret megnövelését is! Mint tudjuk, alapesetben a karakteres kép 80 karakter széles, és a karakterek 9 pixel szélesek. Tehát ezen a képen valójában 720 pixel van vízszintesen. Ha most 8 pixel szélességűre kapcsolnánk a karaktereket, és a kép logikai hosszát 90 karakterre növelnénk, akkor vízszintesen 10 karakterrel több férne el. Sajnos ez még kevés, mert a horizontális CRTC regisztereket is módosítani kell. Az alábbi módosítások szükségesek:

Regiszter	80 karakter	90 karakter
<i>\$3D4,\$00, Horizontal Total</i>	<i>\$5F(95)</i>	<i>\$6B(107)</i>
<i>\$3D4,\$01, Horizontal Display End</i>	<i>\$4F(79)</i>	<i>\$59(89)</i>
<i>\$3D4,\$02, Start Horizontal Blank</i>	<i>\$50(80)</i>	<i>\$5A(90)</i>
<i>\$3D4,\$03, End Horizontal Blank</i>	<i>\$82(130)</i>	<i>\$8E(142)</i>
<i>\$3D4,\$04, Start Horizontal Retrace</i>	<i>\$55(85)</i>	<i>\$5E(94)</i>
<i>\$3D4,\$05, End Horizontal Retrace</i>	<i>\$81(129)</i>	<i>\$8A(138)</i>

A 90 karakteres felbontáshoz tartozó értékeket a 80 karakteres értékek megnövelésével értük el. A beállítást az alábbi eljárás végzi:



NSTDCRT.PAS

```
Procedure To90C(clock:byte);
```

```
Begin
```

```
portw[ $3c2] :=clock; { Az órajel beállítása }
```

```
portw[ $3d4] :=$0e11; { A $3D4,0..7 regiszterek írásának engedélyezése }
```

```
{ A 90 karakteres horizontális CRTC értékek beállítása }
```

```
for i:=0 to 5 do portw[$3d4]:=256*data_360[i]+i;
```

```
portw[ $3d4] :=$2d13; { Egy sor logikai hossza $2d(45) -> 90-et jelent }
```

```
portw[ $3d4] :=$8e11; { A $3D4,0..7 regiszterek írásának letiltása }
```

```
End;
```

Az eljárásban használt *data_360* tömb az előbbi horizontális értékeket tartalmazza. A *clock* paramétert külön kell megadni, mert a *\$3C2* regiszterben a függőleges felbontás is szerepel. Az eljárást a *_90x25*-ös mód beállításához a következőképpen hívjuk meg:

```
Procedure _90x25;
```

```
Begin
```

```
SetCrtMode(_80x25); { A 80x25-ös mód beállítása }
```

```
Box8; { Átváltunk 8 pixel széles karakterekre }
```

```
To90C($67); { Beállítjuk a 90 vízszintes karaktert }
```

```
LogicalSize:=180; { A logikai hossz itt 180 (2*90) }
```

```
End;
```

A két eljárás az **NSTDCRT.PAS** programban található, ahol a többi 90 karakteres mód beállítása is szerepel. Látható, hogy a *Crtplus* egységet is felhasználjuk a mód beállítására. A *LogicalSize* átállításával a *Crtplus* összes eljárása és függvénye a 90 karakter széles módokban is tud dolgozni. A *Crt* egység egyes eljárásai sajnos nem működnek helyesen (pl. a *clrscr*), mert ezek a BIOS funkciókkal kérdezik le a kép szélességét, és ezt mi nem tudjuk beállítani. Mindezek ellenére a mód jól használható.

10.2. Bitplane szervezésű grafikus módok

A 16 színű grafikus módokban a vízszintes és a függőleges felbontások összes lehetséges variációját meg tudjuk valósítani. A lemez mellékleten az **NSTDG16.PAS** programban az összes lehetséges variáció megtalálható. A módszer minden beállítás esetén ugyanaz: beállítjuk valamelyik szabványos módot, és utána vagy a vízszintes, vagy a függőleges méretet változtatjuk meg. Mivel elég sok szabványos mód van, ezért célszerűen ki lehet választani, hogy melyik szabványos módból származtatjuk a nem szabványos módot. A programban a 90 karakteres mód mintájára szerepel két 720 pixeles felbontás is, amelyeket a 640 pixeles felbontás vízszintes méreteinek megváltoztatásával érünk el, így a legnagyobb beállítható felbontás a 720×480-as tizenhat színnel.



NSTDG16.PAS

Az átállításokhoz három táblázatot definiálunk, amelyekkel később a CRTC regisztereket fogjuk módosítani.

```
const
  data_350 : Array[ 0..7]  of
byte=($83,$85,$5D,$28,$0F,$63,$BA,$e3);
  data_480 : Array[ 0..23] of byte=($2D,$27,$28,$90,$2B,$80,$0b,$3e,
                                     $00,$40,$00,$00,$00,$00,$00,$00,
$ea,$8c,$df,$14,$00,$e7,$04,$e3);
  data_720 : Array[ 0..5]  of byte=($6B,$59,$5A,$8E,$5E,$8A);
```

A programban szerepel egy 720×400-as és egy 720×480-as módot beállító eljárás. A 640 pixelről 720 pixelre átkapcsoló eljárás a *To720*, amely a horizontális CRTC regisztereket írja át:

```
Procedure To720(clock:byte); { A 720 pixeles vízszintes felbontás
                                beállítása }
Begin
  portw[ $3c2] :=clock;      { Az órajel beállítása }
  portw[ $3d4] :=$0e11;      { A 0..7 regiszterek írásának engedélye-
                                zése }
  for i:=0 to 5 do          { A horizontális regiszterek átírása }
    portw[ $3d4] :=256*data_720[ i]+i;
  portw[ $3d4] :=$2d13;      { A logikai hossz átírása (90-et jelent) }
  portw[ $3d4] :=$8e11;      { A 0..7 regiszterek írásának tiltása }
End;
```


A későbbi nem szabványos módokat mindig valamelyik szabványos módból származtatjuk, ezért minden szabványos mód beállítására készítünk egy-egy eljárást. Az eljárások a BIOS módbeállító funkcióval bekapcsolják a megadott felbontásokat:

```
{ Szabványosan beállítható módok }
Procedure _320x200; Assembler;
Procedure _640x200; Assembler;
Procedure _640x350; Assembler;
Procedure _640x480; Assembler;
```

A szabványos módok segítségével most minden egyes kombinációt megvalósítunk. Nézzük néhány példát!

```
Procedure _320x400;
Begin
    _320x200;           { A 320x200-as módból indulunk ki           }
    portw[ $3d4] := $4009; { A rasztersorokat megfelezzük           }
                        ($3D4,$09:7.bit)
End;
```

```
Procedure _320x350; Assembler;
Asm
    call _320x200          { A 320x200-as módból indulunk ki           }
    mov  dx,$3C2
    mov  al,$A3             { Az órajel megváltoztatása           }
    out  dx,al
    lea  si,DATA_350        { A data_350 szolgáltatja a CRTC érté- }
                        keket
    mov  cx,8               { Nyolc darab regisztert kell átírni   }
    mov  dx,$3D4
    mov  ax,$4009           { A rasztersorokat megfelezzük           }
    out  dx,ax
    mov  al,$10             { A $10-es regisztertől kezdve módosítunk }
@Set_regs:
    mov  ah,[ si]
    out  dx,ax
    inc  al
    inc  si
    loop @Set_regs
    mov  ax,$1413
    out  dx,ax
End;
```

A 720 pixeles vízszintes felbontást a *To720* eljárással tudjuk beállítani:

```

Procedure _720x400;
  Begin
    _640x400;           {A 640x400-as felbontásból indulunk ki           }
    To720($67);         { A vízszintes felbontást 720-ra állítjuk         }

  End;

Procedure _720x480;
  Begin
    _640x480;
    To720($e7);
  End;

```

A programból itt csak néhány eljárást emeltünk ki, de ezekből jól nyomon követhetők az átkapcsolások. A *Graph* egység rutinjait a 640 pixeles felbontásokban jól lehet használni, csak az eljárások meghívása előtt be kell csapni az egységet. Ha egy *InitGraph* eljárással a *VgaHi*, azaz a 640×480-as felbontást inicializáljuk, és ezek után kapcsoljuk be valamelyik 640 pixel széles felbontást, akkor az összes eljárás jól működik majd. A 320 pixeles felbontásokban alapesetben ezt nem tehetjük meg, mert a képernyő logikai hossza ilyenkor csak 40 bájt. Ha a *\$3D4*, *\$13 Offset* regiszterrel 80-ra állítjuk a logikai hosszúságot (a regiszterbe \$28-at kell írni), akkor itt is tudjuk használni az összes *Graph* eljárást és függvényt. A *Grplus* egységben az *Lsize* értékét kell megváltoztatni, és akkor ezek az eljárások is alkalmazhatók lesznek. A 720 pixeles felbontást a *Graph* egységgel sajnos nem lehet használni, de a *Grplus* egység az *Lsize=90* értékadás után itt is alkalmazható.

10.3. A 256 színű módok

A 256 színű módok közül a 320×200-as *normál bájt/pixel* szervezésű kép szabványos csak, ezért a *módosított bájt/pixel* szerkezetnél ismertetett összes felbontás már nem szabványosnak mondható. Sajnos a 256 színű grafikánál nem tudjuk beállítani a 640 pixeles vízszintes felbontást, itt a legnagyobb lehetséges felbontás a 360×480-as. A beállítás ugyanúgy történik, mint az előző esetekben, a *módosított bájt/pixel* képszerkezet beállítása után a CRTC regiszterek közvetlen átírásával megváltoztatjuk a függőleges felbontást. Ezeket az eljárásokat a *Gr256_2* egység tartalmazza. Hátravan még a 360-as felbontás beállítása, amit a horizontális méretek megváltoztatásával (ugyanúgy, ahogy a 640/720-as váltást tettük) végezhetünk el. A lemezmellékleten található **NSTDG256.PAS** programban láthatjuk a konkrét megvalósítást:



NSTDG256.PAS

```
{ $G+,X+}
```

```
Uses Gr256_2,Crt;
```

```
var i,j,k,l:integer;
```

```
const
```

```
data_360 : Array[ 0..5] of byte=($6B,$59,$5A,$8E,$5E,$8A);
```

```
Lsize = 90;      { A kép logikai hossza } }
```

```
{ Egy pixel kigyújtása a 360 pixel széles 256 színű képernyőn } }
```

```
Procedure Pixel360(X,Y:word;Color:byte); Assembler;
```

```
Asm
```

```
mov ES,SegA000      { Az ES mutat a képernyőre } }
```

```
mov ax,Lsize        { Az Lsize ilyenkor 90 (360/4) } }
```

```
mul Y
```

```
mov di,X
```

```
mov bx,di
```

```
shr di,2
```

```
add di,ax           { A DI-ben megjelenik a képcím } }
```

```
and bx,3            { A BX-szel a pixelpozíciót határozzuk meg } }
```

```
add bx,offset PixelPos
```

```
mov dx,03c4h
```

```
mov al,02h
```

```
mov ah,[ bx]        { A bitmaszk kiírása a $3C4,$02 regisz- terbe } }
```

```
out dx,ax
```

```
mov al,Color
```

```
mov ES:[ di],al     { A pixel kigyújtása } }
```

```
End;
```

```
Procedure To360(clock:byte); { Átkapcsolás a 360 pixeles víz- szintes módba } }
```

```
Begin
```

```
portw[ $3c2] :=clock;    { Az órajel megváltoztatása } }
```

```
portw[ $3d4] :=$0e11;
```

```
for i:=0 to 5 do        { A CRTC horizontális regiszterek át- írása } }
```

```
portw[ $3d4] :=256*data_360[ i] +i;
```

```
portw[ $3d4] :=$2d13;    { A logikai hossz 90 bájt/sor (2d) } }
```

```
portw[ $3d4] :=$8e11;
```

```
End;
```

```
{ A 360-as vízszintes módok beállítása } }
```

```
Procedure _360x200;
```

```
Begin
```

```
_320x200;
```

```
to360($67);
```

```
End;
```

```

Procedure _360x240;
  Begin
    _320x240;
    to360($e7);
  End;
Procedure _360x350;
  Begin
    _320x350;
    to360($a7);
  End;
Procedure _360x400;
  Begin
    _320x400;
    to360($67);
  End;
Procedure _360x480;
  Begin
    _320x480;
    to360($e7);
  End;

BEGIN
  _360x480;
  repeat
    pixel360(random(360), random(480), random(256));
  until keypressed;
  readln;
  textmode(c80);
END.

```

A program elején található *Pixel360* rutin kirajzol egy pixelt a 360 pixel széles képre. A *To360*-nal a vízszintes felbontást módosítjuk. A módbeállító rutinok a *Gr256_2* egységben található *módosított bájt/pixel* formátumú felbontások beállításait felhasználva végzik el a megfelelő felbontás beállítását. Alapesetben csak a *Pixel360* rutin képes a 360 pixeles képekre való rajzolásra, de ha a *Gr256_2* egységben az *Lsize* változót 90-re állítjuk, akkor minden eljárás és rutin használható lesz ezekben a felbontásokban is.

A szabványtól eltérő módok beállításának módszerét ezzel megismertük. Jól látható, hogy a beállítások tulajdonképpen nem igényelnek semmilyen ördögös programozói fogást, csupán tisztában kell lenni a regiszterek programozásának szabályaival, az alkalmazható értékekkel és a regiszterek között fennálló összefüggésekkel. A beállítások sokfélesége is mutatja, hogy mennyi mindent tud a VGA kártya, amelyet a gyártók általában nem — vagy csak jó pénzért — közölnek a programozókkal, és éppen ezért nem vagyunk teljesen tisztában a lehetőségeinkkel.

11. SVGA kártyák, a T8900C SVGA



TRIDENT

A Super VGA kártyákból több típus is forgalomban van. Az SVGA kártyák mindegyike VGA kompatibilis, de a VGA lehetőségein felül még több szöveges és grafikus mód beállítására is lehetőségünk van. Az SVGA kártyáktól az SVGA módokban — alapesetben — ne is várjunk kompatibilitást. Az úgynevezett VESA szabvány segítségével az SVGA kártyák szabványossá tehetők, de ez még mindig csak BIOS szintű kompatibilitást jelent. A VESA szabvány definiál olyan módbeállító és egyéb funkciókat, amelyekkel a normál VGABIOS-hoz hasonlóan INT 10h megszakítással lehet kezelni minden SVGA kártyát. Ezért általában minden SVGA típushoz mellékelnek olyan fontosabb segédprogramokat (drivereket), amelyekkel a legelterjedtebb szoftverek dolgozni tudnak. A felhasználók lemezen kapják meg ezeket a segédprogramokat minimális dokumentációval. Ezekben a dokumentációkban külön a kártya programozására vonatkozóan általában semmiféle információ nem szerepel, legfeljebb a beállítható videomódok funkciókódja. Ebben a könyvben képtelenség volna minden egyes SVGA kártyára kitérni, hiszen mindegyik kártyáról önálló könyvet lehetne írni. Ezért a Magyarországon forgalomban lévő kártyák közül most csak a **Trident 8900C** típusjelű SVGA kártyával fogunk részletesebben foglalkozni. Ez a kártya viszonylag olcsó, elég sok jó tulajdonsága van és eléggé elterjedt. A továbbiakban tehát minden információ erre az SVGA kártyára vonatkozik, ezért más típusok esetén a közölt programok nem alkalmazhatók.

11.1. Trident SVGA videomódok

A kártyán minden egyes szabványos és nem szabványos VGA mód beállítható, amelyekről az előző fejezetekben szó volt, azokon felül pedig a következő módokat ismeri még:

Mód	Típus	Felbontás	Kezdőcím	Karakterméret
\$50	text	80×30	B800:0000	8×16
\$51	text	80×43	B800:0000	8×11
\$52	text	80×60	B800:0000	8×8
\$53	text	132×25	B800:0000	8×14
\$54	text	132×30	B800:0000	8×16

Mód	Típus	Felbontás	Kezdőcím	Karakterméret
\$55	text	132×43	B800:0000	8×11
\$56	text	132×60	B800:0000	8×8
\$57	text	132×25	B800:0000	9×14
\$58	text	132×30	B800:0000	9×16
\$59	text	132×43	B800:0000	9×11
\$5a	text	132×60	B800:0000	9×8
\$5b	graph	800×600/c16	A000:0000	(8×8)
\$5c	graph	640×400/c256	A000:0000	(8×16)
\$5d	graph	640×480/c256	A000:0000	(8×16)
\$5e	graph	800×600/c256	A000:0000	(8×8)
\$5f	graph	1024×768/c16	A000:0000	(8×16)
\$60	graph	1024×768/c4	A000:0000	(8×16)
\$61	graph	768×1024/c16	A000:0000	(8×16)
\$62	graph	1024×768/c256	A000:0000	(8×16)

Egy ilyen táblázatot általában minden egyes SVGA kártyához ad a gyártó, így ezeket a módokat a BIOS videomód-beállító funkciójával beállíthatjuk. Azt, hogy a fenti módokból melyik állítható be, mindig a kártya saját memóriája szabja meg. A legtöbb kártya 512K és 1M memóriával kapható, bár működik 256K-val is, de a kereskedelemben így általában nem tudjuk megvenni. Ha a kártyán 512K memória van, akkor a legutolsó módot nem tudjuk beállítani (1024×768/c256).

11.2. Szöveges módok

A szöveges módok képszerkezete megegyezik a normál VGA karakteres képszerkezettel, ezért ezekre most nem is térünk ki részletesen. Ha a CRT egységet használjuk, akkor bármelyik SVGA text mód alkalmazható, és csupán az ablakdefiniáló *window* eljárás helyett kell egy másik eljárást írni. Ezt is csak azért, mert a CRT egység *windmax* változójának beállítását ez 80-nál nagyobbra nem engedi. A *window* eljárást az alábbi eljárással helyettesíthetjük:

```

Procedure Twindow(X1,Y1,X2,Y2:integer);
Begin
    windmin:=(X1-1)+256*(Y1-1);
    windmax:=(X2-1)+256*(Y2-1);
End;

```

Az eljárás helyettesíti a *window* eljárást, és a nagyobb, 132 karakteres módokban is tudjuk használni az összes CRT rutint. Természetesen ilyenkor egy karaktersor logikai hossza 132*2 bájt, és ezt minden esetben figyelembe kell venni.

11.3. Bitplane szervezésű grafikus módok

Két új nagy felbontású 16 színű módunk van: 800×600 (\$5B) és 1024×768 (\$5f). Ezek a módok bitplane képszervezésűek, ezért a programozásuk is ugyanúgy történik, mint a szabványos bitplane szerkezetű képeké. Sajnos az 1024×768-as felbontás esetében a 64K rendelkezésre álló címtartomány nem elegendő a teljes képernyőhöz, ezért itt az úgynevezett memóriabank-technikát kell alkalmazni, amit majd a 256 színű nagy felbontású módoknál ismerhetünk meg.

11.4. Bájt/pixel szervezésű grafikus módok

A T8900C a következő 256 színű felbontásokat tudja: 640×400, 640×480, 800×600, 1024×768 (1 Mbájt memóriával). Ezek a módok is egyszerűen a BIOS meghívásával állíthatók be (INT 10h).

Az SVGA kártyák esetében a VGA kártyákhoz képest az a legfontosabb előrelépés, hogy az SVGA-knak nagy felbontású 256 színű videomódjaik vannak. Ezek a módok minden esetben *normál bájt/pixel* képszervezésűek, de a címterület nagysága miatt használatuk a különleges, bank lapozási technikát igényli.

11.4.1. A bank lapozási technika

Vegyük alapul a 640×480-as módot 256 színnel! Ennek a memóriaigénye 640*480, tehát 307 200 bájt. Világos, hogy ez a terület nem fér bele az A000:0000 címen fenntartott 64K-s képszegmensbe, de még a 128K-s maximális videoterületre sem. Ezért az SVGA kártyákon ilyen esetekben csak lapozással tudjuk elérni a teljes képmemóriát. A kártya videomemóriája úgynevezett *bankokból* (lapokból) áll, amelyeknek a hossza 64K. Az installált memória nagyságától függően a kártyán lehet négy (256K), nyolc (512K) vagy tizenhat (1M) lap. Amikor bekapcsolunk egy nagy felbontású 256 színű grafikus módot, akkor a felbontástól függően több ilyen lapon látható a kép, de az A000 szegmensben ténylegesen csak egy lapot lehet elérni. A 640×480-as kép esetében 4 teljes lap látható és egy, amelynek a vége már nem jelenik meg ($307\,200/65\,536=4.6875$). Minden SVGA-nak van egy lapozóregisztere, amellyel meg lehet adni az aktuális lap számát. A Trident esetében ez a \$3C4, \$0E *Bank Select* (lapkiválasztás) regiszter, amelynek az alsó 4 bitjén kell megadni az aktuális lap számát. Valamilyen oknál fogva a lap számának 1. bitje invertálva van, ezért a lap kijelölésekor az 1. biten invertált számot kell kiírni a regiszterbe a tényleges lapszám

helyett. A felső biteknek is van valamilyen — nem dokumentált — szerepük, ezért oda minden esetben 0111b értéket kell kiírni. Mindezeket figyelembe véve írjunk egy olyan lapkiválasztó eljárást, ahol mi adjuk meg az aktuális lapot:



BANKS.PAS

```
Uses Crt;
```

```
var i : word;
```

```
Procedure BankSelect (B:byte);
```

```
Begin
```

```
  b:=b and $0f;           { Csak az alsó négy bitet tartjuk meg }
  b:=b xor $72;           { Az 1. bit invertálása, a felső bitekbe
```

```
                        0111b }
```

```
  portw[ $3c4] := (b shl 8) + $0E; { Bankkiválasztás }
```

```
End;
```

```
BEGIN
```

```
  asm
```

```
    mov  ax, $5D
```

```
    int  10h           { 640x480/c256 grafikus mód }
```

```
  end;
```

```
  for i:=0 to 4 do
```

```
    Begin
```

```
      BankSelect(i);           { A következő bank kiválasztása }
```

```
      Fillchar(mem[ $a000:0], 65535, i+1); { Feltöltés egy színnel }
```

```
      mem[ $a000:$ffff] :=15;    { A bank legutolsó pontja fehér
                                lesz }
```

```
    End;
```

```
  readln;
```

```
  textmode(c80);
```

```
END.
```

A program futtatásakor különböző színekben megjelennek a lapok, a határokat pedig egy-egy fehér pixellel jelöljük. A program alapján láthatjuk a kép külön 64K-s mezőit, amelyeket csak egyenként tudunk elérni. Mivel a képszerkezet *normál bájt/pixel* formátumú, ezért az összes rutin alkalmazható egy lapon belül, amit a *GR256.PAS* programban is megírtunk.

A legtöbb SVGA kártya tulajdonképpen a laptechnikát alkalmazza, ezért a fennálló inkompatibilitás ellenére is lehet olyan programokat írni, amelyek azután minimális módosításokkal más-más SVGA kártyákon is futtathatók. A programok átviteléhez két dolgot kell tudni:

1. A beállítás módját, a létező videomódok kódját. A fenti négy 256 színű felbontás szinte minden SVGA kártyán beállítható, csak a kódok mások. A kártyához mellékelt rövid ismertető füzetben általában szerepelnek a módok és a kódjaik.
2. A banklapozás regiszterét és a lapozás technikáját. Ezt az információt sajnos csak közvetlen nyomkövetéssel lehet megszerezni. Írni kell egy olyan programot, amely a BIOS pixelrajzoló (\$0C) funkciójával rajzol a képre két-három pixelt, amelyek biztosan külön lapon vannak. Ezeket a rutinokat sorra nyomon kell követni, és fel kell jegyezni minden egyes I/O műveletet. Ha például a 640×480-as képre kirakunk három pontot (egyet a 0,0, egyet a 320,240 és egyet a 639,479 pozícióba), akkor a pontok kirakásakor a BIOS biztosan lapot vált. Ezért az összes I/O művelet közül általában minden SVGA esetben ki lehet szűrni a lap váltásának algoritmusát. A módszer egy kissé időigényes és némi gyakorlatot igényel, de általában megtérül a befektetett munka.

11.4.2. A T8900C egység

A lemez mellékleten található *T8900C.PAS* egy olyan unit forrásprogramja, amellyel a Trident kártya nagy felbontású 256 színű módjait tudjuk használni. Az egységben a korábbi fejezetekben megismert eljárások szerepelnek, azaz tulajdonképpen a *GR256.PAS* egység rutinjai, kiegészítve a banklapozás miatt szükséges néhány módosítással. Az elvi megoldások természetesen ugyanazok, csak itt minden egyes képmemória írás után meg kell néznünk, hogy nem léptünk-e át lap határt, és ebben az esetben lapozni kell. Ezért a forrásprogramokat most nem tárgyaljuk, hanem csak az egység által nyújtott szolgáltatásokat vesszük sorra.

```
{ Színek a rajzoláshoz és a szövegkiíráshoz }
Ink   : byte = 15;
Paper : byte = 01;
Over  : byte = 01;

{ Színdefiníciók a keretezett téglalapokhoz (BBAR,CBAR) }
Cols  : Array[ 0..11] of byte=(15,8,7,8,15,7,15,8,3,8,15,3);

Vspace : word = 8;      { A betűtávolság szövegkiíráskor }
```

A színezés a *GR256* egységnél megismertek szerint az *Ink*, a *Paper* és az *Over* változók beállításával lehetséges. Az összes rutin az *Ink* színnel rajzol.

Eljárások és függvények:

Procedure InitScreen (Mode:byte);

A videomód beállítása. Az eljárás beállítja valamelyik 256 színű TVGA módot. Az alábbi előre definiált állandók használhatók:


```

_640x400 = $5C;
_640x480 = $5D;
_800x600 = $5E;
_1024x768 = $62;

```

A videomód beállítása mellett az eljárás elvégez még néhány inicializálást, ezért minden beállításkor célszerű használni.

```

Procedure SetBank;
Procedure NextBank;
Procedure PrevBank;

```

Lapváltás. A *SetBank* az AH számú lapot állítja be, a *NextBank* a következő lapot, a *PrevBank* pedig az előző lapot. Az eljárásokat más rutinok használják, de az esetleges külső rutinok számára is elérhetők. Az egység deklarál egy *Bank* változót, amely mindig az aktuális beállított lap számát tartalmazza. A *SetBank* az AH értékét teszi ebbe a változóba, míg a *NextBank* és a *PrevBank* 4 biten lépteti (inc/dec) a változót.

```

Procedure PutPixel(X,Y:word;Color:byte);

```

Egy pixel kigyújtása. Az eljárás *Color* színűre változtatja az *X,Y* koordinátájú pixelt.

```

Function GetPixel(X,Y:word):byte;

```

Egy pixel színének lekérdezése. A függvény visszaadja az *X,Y* koordinátájú képpont színét.

```

Procedure Line(x1,y1,x2,y2:word);

```

Általános egyenes rajzolása. Az eljárás egyenest húz *Ink* színnel az *X1,Y1* és az *X2,Y2* koordinátákkal megadott két pont között.

```

Procedure VLine(X,Y,Long:word);

```

Függőleges egyenes rajzolása. Az eljárás *Long* hosszúságú függőleges egyenest húz az *X,Y* kezdőpontból. Az *X,Y* mindig a felső kezdőpontot jelöli, és a *Long* hosszúság nem lehet nulla. A rajzolás *Ink* színnel történik.

```

Procedure HLine(X,Y,Long:word);

```

Vízszintes egyenes rajzolása. Az eljárás *Long* hosszúságú vízszintes egyenest húz az *X,Y* kezdőpontból. Az *X,Y* mindig a bal szélső kezdőpontot jelöli, és a *Long* hosszúság nem lehet nulla. A rajzolás *Ink* színnel történik.

Procedure Box (X1,Y1,X2,Y2:word);

Üres téglalap rajzolása. Az eljárás az *X1,Y1* és az *X2,Y2* koordinátákkal megadott két átlós végpont által meghatározott üres téglalapot rajzol. A rajzolás *Ink* színnel történik.

Procedure Bar (x1,y1,x2,y2:word);

Kitöltött téglalap rajzolása. Az eljárás az *X1,Y1* és az *X2,Y2* koordinátákkal megadott két átlós végpont által meghatározott kitöltött téglalapot rajzol. A rajzolás *Ink* színnel történik.

Procedure Bbar (X1,Y1,X2,Y2,Ty:word);

Procedure Cbar (X1,Y1,X2,Y2,Ty:word);

Keretezett téglalapok rajzolása. Az eljárások keretezett téglalapokat rajzolnak a *Gr256.PAS* programnál leírt színdefiníciók és szabályok szerint.

Procedure OutCharXY (X,Y:word;C:char);

Egy karakter kiírása. Az *X,Y* pozícióba 8×14-es méretben kiírja a *C* kódú karaktert *Ink* színnel. A háttér színezésére a *GR256.PAS* programnál leírtak érvényesek.

Procedure OutStrXY (X,Y:word; S:string);

Egy szöveg kiírása. Az *X,Y* pozícióba kiírja az *S* szöveget. A színezés a *GR256.PAS* programnál leírtak szerint alakul.

Function ImageSize (X1,Y1,X2,Y2:word):longint;

Egy ablak memóriaigényének meghatározása. A függvény visszaadja az *X1,Y1* és az *X2,Y2* koordinátákkal megadott téglalap memóriaigényét a *GetImage* számára. A függvény visszatérési értéke *longint* típusú.

Procedure GetImage (X1,Y1,X2,Y2:word; var Dest);

Egy ablak eltárolása a memóriában. Az eljárás az *X1,Y1* és az *X2,Y2* koordinátákkal megadott ablak tartalmát a *Dest* típus nélküli változó által meghatározott memóriacímre menti. A *Dest* tipikusan egy lefoglalt heapterületre mutató pointer. Az ablak méretét az *ImageSize* függvénnyel állapíthatjuk meg. Ha az ablak 64K-nál több területet igényel, akkor több kisebb területként kell elmenteni.

Procedure PutImage(X,Y:word; **var** Source);

Egy eltárolt ablak kirajzolása az X,Y koordinátába. Az eljárás a *GetImage* eljárással elmentett területet visszatölti a képre az X,Y koordinátákkal megadott helyre. A *Source* az elmentett terület címét határozza meg.

Procedure ClearTridentMem;

A videomemória törlése. Az eljárás törli a Trident 16 lapját, azaz nullával tölti fel ezeket. 512K-s kártyák esetén a lapszámot 8-ra kell módosítani a forráslistában.

Procedure SetRGBreg(Regnum, Red, Green, Blue:byte);

Procedure GetRGBreg(Regnum:byte; **var** RGB);

RGB regiszterek beállítása és lekérdezése. A rutinok működését a *GR256.PAS* és a *GR256_2.PAS* egységekben részletesen ismertettük.

Sajnos a könyv terjedelme nem teszi lehetővé, hogy ennél részletesebben foglalkozunk az SVGA kártyákkal. Ezek a grafikai eljárások csupán az alapfeladatokat oldják meg, a feladatok nagysága jól érzékelhető az eljárások forrásprogramjainak megtekintésekor. Befejezésképpen a VESA szabvány egyik lehetőségének bemutatására futtassuk le a *HISCROLL.PAS* programot! Egyes Trident kártyák alapesetben is felismerik a VESA-BIOS megszakításait, másoknál viszont szükség lehet a *VESA.EXE* (Trident gyári lemezmelléklet) program elindítására. A program a 800×600-as 256 színű képet görgeti pixelesen. A program 1M memóriával felszerelt kártyákon fut csak.

Tárgymutató

16 színű BOB 192, 215
16 színű grafikus mód 24, 30, 33, 71, 129,
141, 215, 227, 260, 287, 323, 327, 335
24 bites kép 30
256 színű BOB 261, 300
256 színű grafikus mód 28, 237, 271, 329,
335

A

ablakkép 93, 218, 221, 314
ablakkezelés 114
active page 56
AdA BOB-mező 182, 205, 302
AdB BOB-mező 182, 205, 302
additív színkeverés 14, 123
aktív kép 65, 180
Amiga 15, 25, 173
AndPut 238, 256
animáció 30, 83, 121, 131, 141, 215, 300
ASCII kód 58, 99, 106
Atari 25
átfedő BOB 141, 203, 261
attr 57, 114
AttrChange 114, 116
Attribute regiszter 35, 76, 92, 225, 322
Attribute Controller 34
attribútum 19
axonometrikus ábrázolás 136
axonometrikus transzformáció 136, 137

B

BACK 152, 238, 248, 274
bájt címezsmód 47, 277
BANK 335
BANK lapozás 335, 337

BANK technika 335, 338
BankSelect 335
Bar 128, 245, 251, 273, 287, 339
Bbar 154, 172, 239, 274, 249, 290, 339
billentyűzetmegszakítás 99, 114, 226, 318
BINOBJ segédprogram 109, 155, 251
BinStr 114, 116
BIOS 16, 24, 31, 49, 237, 333
BIOS funkció 55..68, 76, 99, 141, 237, 271
bitmaszk 44, 50, 143, 152, 192, 277, 301
bitminta 78, 156
bitplane 24, 35, 39, 42, 44, 49..53, 79, 141,
204, 237, 327, 335
bitplane írásmaszk 27, 39, 79
bitplane olvasásmaszk 79
bitplane szervezésű képszerkezet 24, 218,
248, 314, 327, 335
blank 13, 44..48, 326
blinking 59, 75
Blitter 172
BOB 141, 172..217, 261, 300
BOB-maszk 203, 204
BOBsize 183, 205, 303
BOBvar 175, 182, 195, 205
border 57, 59, 114
Box 130, 154, 239, 274, 339
Box8 73, 114
Box9 74, 114
buffer kép 180, 261..269, 300
BuffSEG 303

C

Cadr 238, 251, 273, 294
Cbar 114, 116
Cbit 273, 294
CGA 15, 26, 39, 47, 58, 64
char 57, 69..76

Char8x8 238, 251, 273, 294
CharChange 114, 116
CharEdit segédprogram 121
CHARMODE 69
Charset 155, 239, 276
ClearScreen 155, 239, 260
ClearSegA000 275, 299
ClearTridentMem 340
CO80 70
Convert_Datas 197, 207, 304
CopyPut 238, 256
CRT egység 70, 113, 326
CRTC 44, 321
CrtPlus egység 113, 326
Cursor 44, 56, 114
CursorOFF 114, 116
CursorON 114, 116

D

DARK 152, 238, 248, 274
DefChar 81, 114, 117
DefChars 77, 114, 117
DMA 19, 25, 173
Double-Buffer 181
DrawBorder 114, 117
dupla rastersoros üzemmód 46, 94
dupla szó címezsmód 47, 277

E

EGA 15, 26, 34, 59, 64
egér 173
egyszínű BOB 141, 182
elektronágyú 11
elektronsugárcső 11
első bájtmaszkok 166, 285
elsődleges (EGA) paletta 34, 59, 120
eltérítő tekercsek 11
Enable128K 233
EQ 114, 117

F

felbontás 12, 14, 23, 64, 321
felhasználói karakterkészlet 62, 76
félképváltás 12
Flash 114, 118
folyamatos képfrissítés 261
Font8x8 70
fordítási direktíva 38, 140
forgató buffer 161, 176, 186
FrameSize 114, 118

G

G16MODE 141
G256MODE 237, 271
GetBIGImage 231
GetFrame 114, 118
GetImage 228, 255, /275, 291, 339
Getpixel 145, 168, 241, 279, 338
GetPlane 230
GetRGBreg 275, 299, 340
globális kép 92..105, 221, 314
görgetés 87, 91, 218, 221, 259, 314
Gr256 egység 127, 238
Gr256_2 egység 273
grafikus kép 22, 24, 28, 30, 141, 237, 271, 327, 335
grafikus mód 35, 141, 237, 271, 327, 335
grafikus objektum 141, 172
Graph egység 26, 146, 150
gray-scale 62
GrPlus egység 150..168

H

háromszög arányosság 126
háttérszín 16, 20, 35, 57
heap 140, 227, 255, 290
Hercules 23, 43, 47
HexStr 114, 118

Hline 154, 166, 274, 285, 338
horizontal blanking 13, 44, 45, 326
horizontális eltolás 35, 218
horizontális eltolás regiszter 36, 92, 222, 314
horizontális órajel 37, 72, 221
horizontális visszafutás 13, 48, 225

I

időzítő frekvencia 112
időzítő megszakítás 84, 111, 120, 216
Image 172
ImageSize 227, 255, 293, 339
impulzus-vezérelt regiszter 191
InacSCR 183, 206, 303
InacSEG 303
inaktív kép 181
INK 151, 238, 273, 337
InstallKeyIRQ 114, 118
intensity 59
Ir BOB-mező 182, 205, 302
írás mód 43, 50..53
IRQ 83, 99, 114

K

karakter 19, 69
karakter bittérkép 30
karakterbox 30
karakteres kép 19, 69, 324, 334
karakteres képszerkezet 19, 334
karaktergenerátor 62
képbájt 26, 49..53
képcím számítás 19, 87, 142, 240, 277
képmemória 19, 49, 227, 255, 290, 335
képmemória másolás 227, 255, 290
képmemória tárolás 227, 255, 290, 339
képpont 14, 58, 142, 240, 277
képváltási frekvencia 13
keretezett téglalapok lásd: Bbar, Cbar

kettősbuffer-technika 180, 300
KeyIRQ 96, 99
kitöltött téglalap lásd Bar

L

labirintus 106, 110
LIGHT 152, 238, 248, 274
Line 146, 241, 279, 338
Line Compare regiszter 48, 102, 222
lineáris színkeverés 126
LoadPicture 235
Logical Size 47
logikai szélesség 142
Lp BOB-mező 182, 205, 302
Lsize 151, 239, 273

M

megszakítás 83, 99, 271
mem[] 20
memória mód 39
módosított bájt/pixel képszerkezet 237, 271
monitorpont 13
monokrom kép 11, 22, 43, 47
MoveBOBs 178, 188, 201, 267, 311
mozgásfázisok 83, 215
mozgató algoritmus lásd: MoveBOBs

N

NE 114, 119
nem átfedő BOB 175, 192
NewWindow 114, 119
NextBank 338
NONSTD 321
normál bájt/pixel képszerkezet 28, 237
NormalPut 238, 256
NotPut 238, 256

O

OAK VGA 38
Offset regiszter 47
olvasás mód 43, 49
órajel 37, 38, 72, 221
OrPut 238, 256
osztott képernyő 102, 221
OutCharXY 153, 158, 161, 253, 297
OutStrXY 158, 254, 297
OVER 152, 238, 273, 337
Overflow regiszter 45

P

Palanim 131
paletta 26, 123
paletta animáció 131, 136
PAPER 151, 238, 273, 337
Paradise VGA 38
pepitamintás fill 129
pixel lásd: képpont
pixel kigyújtása 142, 240, 271, 338
pixel lekérdezése 144, 241, 279, 338
pixeltorzulás 14, 190
pixeltorzulási hányados 190
plane lásd: bitplane
PlaneSize 229
Plot 132
port[] 33
portw[] 33
Preset Row Scan regiszter 45
PrevBank 338
prioritás 173, 175, 182
programozott mozgás 193, 201
PutBIGimage 232
PutFrame 114, 119
PutImage 227, 239, 256, 275, 292
Putpixel 142, 240, 277, 338
PutPlane 231

R

rasztersor 13
rasztersor számláló 46
Rbuff 160, 176, 184, 206
Read Plane select regiszter 42
Realtek VGA 38
regiszter 33
RestoreImage 234
retrace 13, 45, 46
RGB 14, 40, 60, 61, 123
RGBI 15

S

SavePicture 235
SCAN kód 99
Scan Line 13, 46
ScreenInit 260, 337
Scroll 87, 218, 221, 259, 314
ScrollDn 114, 119
ScrollLf 114, 119, 219
ScrollRg 114, 119
ScrollUp 88, 114, 119, 218
SegA000 238, 273
Sequencer regiszter 37..39
sequencer reset 38
SetBank 338
SetCrtMode 114, 119
SetCursorSize 114, 120
SetEGAPalette 114, 120
Setpalette 26
SetRGB 114, 120
SetRGBpalette 26
SetRGBreg 245, 298, 340
SetTimer 114, 120
Shape 173
ShowBOBs 179, 185, 199, 208, 307
ShowPIC 93, 222, 317
SolidBar 154, 169

splitscreen 102
Sprite 173
Start Address regiszter 46
Státus regiszter 36
StoreImage 233
StrBin 114, 120
StrHex 114, 120
Strobe 191
SVGA 333
szabványos képfelbontások 55, 69, 141, 237, 323, 333
szellemkép 181
színbájt 20
színelmélet 13, 123
színes bájt 26, 33, 43, 49..53
színkeverés 14, 125
színkomponens 15, 40, 60, 61
szó címzés mód 19, 47, 48
szögfüggvényes színkeverés 128
szubsztraktív színkeverés 14, 123

T

takarás 139, 175, 181
Textmode 70
TimerIRQ 84, 217
többszínű BOB 192, 203
Trident VGA 38, 333
True Color 30, 125
Twindow 334
Ty paraméter 154

U

Underline 47
UninstallKeyIRQ 114, 120
utolsó bájtmaszkok 166, 285

Ü

üres intervallum 13
üres téglalap 154, 239, 273, 339

V

vastagon keretezett téglalap lásd: Bbar
vékonyan keretezett téglalap lásd: Cbar
véletlenszerű mozgás 188, 205
vertical blanking 13, 45, 47
vertikális eltolás 45, 222
vertikális órajel 37, 72
vertikális visszafutás 13, 46, 47
VESA 333, 340
VGA latch 33, 49..53
VGA-BIOS 55
video-sequencer 37..39
videociklus 38
videomemória 19, 43, 49..53
videomód 55
villogás 20, 35, 59
villogás bit 75
visszafutás 13, 46
Vline 153, 163, 274, 284, 338
vonalarajzolás lásd: Line, Hline, Vline
Vspace 238, 252, 254, 337

W

windmax 70, 334
windmin 334
window 57, 114, 119, 334
Write Plane Enable regiszter 39
WriteMode lásd: írás mód

X

XorPut 238, 256

Táblázatok

Színtáblázatok

Fontosabb színátmenetek:

fekete-kék

R	G	B
00h	00h	08h
00h	00h	10h
00h	00h	18h
00h	00h	20h
00h	00h	28h
00h	00h	30h
00h	00h	38h
00h	00h	3Fh

fekete-zöld

R	G	B
00h	08h	00h
00h	10h	00h
00h	18h	00h
00h	20h	00h
00h	28h	00h
00h	30h	00h
00h	38h	00h
00h	3Fh	00h

fekete-ciánkék

R	G	B
00h	08h	08h
00h	10h	10h
00h	18h	18h
00h	20h	20h
00h	28h	28h
00h	30h	30h
00h	38h	38h
00h	3Fh	3Fh

fekete-piros

R	G	B
08h	00h	00h
10h	00h	00h
18h	00h	00h
20h	00h	00h
28h	00h	00h
30h	00h	00h
38h	00h	00h
3Fh	00h	00h

fekete-lila

R	G	B
08h	00h	08h
10h	00h	10h
18h	00h	18h
20h	00h	20h
28h	00h	28h
30h	00h	30h
38h	00h	38h
3Fh	00h	3Fh

fekete-sárga

R	G	B
08h	08h	00h
10h	10h	00h
18h	18h	00h
20h	20h	00h
28h	28h	00h
30h	30h	00h
38h	38h	00h
3Fh	3Fh	00h

fekete-fehér

R	G	B
08h	08h	08h
10h	10h	10h
18h	18h	18h
20h	20h	20h
28h	28h	28h
30h	30h	30h
38h	38h	38h
3Fh	3Fh	3Fh

fekete-barna

R	G	B
07h	02h	00h
08h	05h	00h
0Ah	05h	00h
14h	0Ah	00h
1Eh	0Fh	00h
28h	14h	00h
32h	19h	00h
3Ch	1Eh	00h

Az alapbeállítású VGA-paletta első tizenhat színe:

R	G	B		R	G	B	
00h	00h	00h	fekete	15h	15h	15h	sötétszürke
00h	00h	2Ah	kék	15h	15h	3Fh	világoskék
00h	2Ah	00h	zöld	15h	3Fh	15h	világoszöld
00h	2Ah	2Ah	ciánkék	15h	3Fh	3Fh	világoscián
2Ah	00h	00h	piros	3Fh	15h	15h	világospiros
2Ah	00h	2Ah	lila	3Fh	15h	3Fh	világoslila
2Ah	15h	00h	barna	3Fh	3Fh	15h	sárga
2Ah	2Ah	2Ah	szürke	3Fh	3Fh	3Fh	fehér

Felbontások és memóriaigények:

Felbontás és színek		CPU látszólagos hossz	Valódi memóriaigény	Képszerkezet
<i>vízszxfügg/szín</i>	<i>txt/grp</i>	(bájt)	(bájt)	
Normál VGA:				
40•25/16	T	2000	2000	word/char
80•25/16	T	4000	4000	word/char
40•50/16	T	4000	4000	word/char
80•50/16	T	8000	8000	word/char
320•200/2	G	8000	8000	1 bitplane
320•200/4	G	16000	16000	CGA
				2bit/pixel
320•200/16	G	8000	32000	4 bitplane
320•200/256	G	64000	64000	bájt/pixel
640•200/2	G	16000	16000	1 bitplane
640•200/16	G	16000	64000	4 bitplane
640•350/2	G	28000	28000	1 bitplane
640•350/16	G	28000	112000	4 bitplane
640•480/2	G	38400	38400	1 bitplane
640•480/16	G	38400	153600	4 bitplane

Felbontás és színek		CPU látszólagos hossz	Valódi memóriaigény	Képszerkezet
<i>vízszxfügg/szín</i>	<i>txt/grp</i>	<i>(bájt)</i>	<i>(bájt)</i>	
SVGA:				
80•30/16	T	4800	4800	word/char
80•43/16	T	6880	6880	word/char
80•60/16	T	9600	9600	word/char
132•25/16	T	6600	6600	word/char
132•30/16	T	7920	7920	word/char
132•43/16	T	11352	11352	word/char
132•60/16	T	15840	15840	word/char
800•600/16	G	60000	240000	4 bitplane
1024•768/16	G	2 bank	393216	4 bitplane
640•400/256	G	4 bank	256000	bájt/pixel
640•480/256	G	5 bank	307200	bájt/pixel
800•600/256	G	8 bank	480000	bájt/pixel
1024•768/256	G	12 bank	786432	bájt/pixel
640•480/16M	G,true-color	15 bank	921600	24bit/pixel
800•600/16M	G,true-color	22 bank	1440000	24bit/pixel
1024•768/16M	G,true-color	36 bank	2359296	24bit/pixel

Kép- és sorfrekvenciák

Mód	Órajel	Sorfrekvencia	Képfrekvencia	Polaritások
<i>(hex)</i>	<i>(MHz)</i>	<i>(KHz)</i>	<i>(Hz)</i>	<i>(V,H)</i>
0,1	28.32	31.5	70	+,-
2,3	28.32	31.5	70	+,-
4,5	25.17	31.4	70	+,-
6	25.17	31.4	70	+,-
7	28.32	31.5	70	+,+
0D,0E	25.17	31.4	70	+,-
0F,10	25.17	31.4	70	-,+
11,12	25.17	31.4	60	-,-
13	25.17	31.4	70	+,-

VESA SuperVGA videomódok

A VESA módok beállítása:

ax = 4f02h

bx = mód (ha a 15.bit 1, akkor nincs képernyőtörlés)

int 10h

Visszatérés: al = 4fh, ha a funkció létezik

ah = 00h - sikeres beállítás

01h - sikertelen beállítás

<u>Mód</u>	<u>Felbontás és szín</u>	<u>SVGA memóriaigény</u>
<i>(hex)</i>	<i>(vízszfügg/szín)</i>	<i>(txt/grp)</i>
0100h	640x400/256	G 512K
0101h	640x480/256	G 512K
0102h	800x600/16	G 512K
0103h	800x600/256	G 512K
0104h	1024x768/16	G 512K
0105h	1024x768/256	G 1M
0106h	1280x1024/16	G 1M
0107h	1280x1024/256	G 2M
0108h	80x60/16	T 512K
0109h	132x25/16	T 512K
010ah	132x43/16	T 512K
010bh	132x50/16	T 512K
010ch	132x60/16	T 512K
010dh	320x200/32K	G,hi-color 512K
010eh	320x200/64K	G,hi-color 512K
010fh	320x200/16M	G,true-color 512K
0110h	640x480/32K	G,hi-color 1M
0111h	640x480/64K	G,hi-color 1M
0112h	640x480/16M	G,true-color 1M
0113h	800x600/32K	G,hi-color 1M
0114h	800x600/64K	G,hi-color 1M

Irodalomjegyzék

IBM Enhanced Graphic Adapter . IBM Personal Computer Hardware Reference Library

Motorola 8 Bit Peripheral Devices. Motorola.

TVGA 8900C, 8900CL, 8900D Card User's Manual. Trident Microsystems, 1992.

Amiga 1200, AGA Supplement User's Guide. Commodore, 1992.

Turbo Assembler Quick Reference Guide. Borland International, 1992.

Abonyi Zsolt:

PC hardver kézikönyv. ComputerBooks, Budapest, 1993.

Angster Erzsébet - Kertész László:

Turbo Pascal 6.0. Budapest, 1992.

Dr. Kovács Magda:

32 bites mikroprocesszorok, 80386 I-II. LSI oktatóközpont, Budapest, 1991.

Drótos Dániel:

Borland Pascal 7.0 . LSI oktatóközpont, Budapest, 1993.

A nyomást és kötést
az Akadémiai Nyomda készítette. Táskaszám: 1107
Felelős vezető: Freier László

DIGITALIZÁLÓ TÁBLÁK MÁR 14 900 Ft-TÓL !, TÉRBELI DIGITALIZÁLÓK

Autodesk szoftverakció AutoCAD és 3D STUDIO ajándék digitalizáló táblával



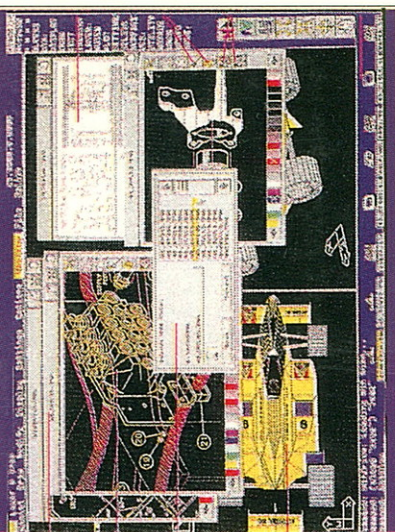
Acecat™ 5" x 5" tablet

Ölébe venne Ön egy egeret?

De! . . . egy tenyérnyi méretű digitalizáló táblán úgy, ahogy tollat vagy ecsetet tart az ujjai között, megszokott számítógépét pen computerré alakítva dolgozhat. Mouse helyett CAT, hogy munkája könnyebben egy más minőség legyen. Az ACECAT kézzel fogható és abszolút precizitást nyújt minden DOS, MS WINDOWS, protected és real módú AutoCAD és MAC alkalmazásban.

ACECAD digitalizáló táblák 12"x12" és 12"x18"-os méretben

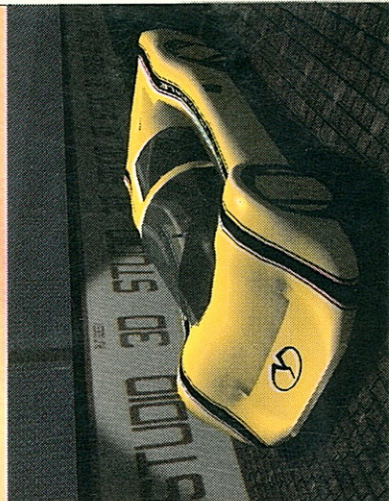
Az abszolút pozicionálás biztosítja a pontosságot és gyorsaságot, hogy alkotó munkáját rekordidő alatt végezhesse el,... könnyedén. Nélkülözhetetlen építészek, kiadványszerkesztők, mérnökök és grafikus tervezők számára. Két- vagy háromgombos ceruzával, 4 vagy 16 gombos célkereszttel.



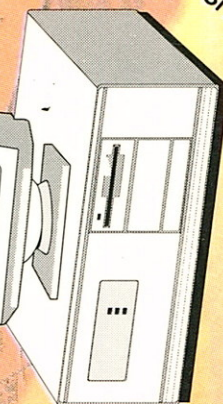
Nth® Graphics Texas

Vektoros monitorvezérlés bármilyen displaykártyával többszörös sebességgel AutoCAD-hez

- rendkívüli sebesség
- real time zoom-pan, birdseye
- kiterjesztett garancia, és update
- a leggyorsabb
- DLP programmal, az **Nth Drive**-val
- a leghatékonyabb
- a legkényelmesebb kezelhető
- animációs programmal, a **Hydra**-val.



digitizáló tábla
Hogy számítógépéből
valódi grafikus
munkahely legyen



síkplotterek
CAD - grafikus szoftver

AUTODESK AutoCAD
3D Studio

ACECAD Digitalizáló táblák
kizárólagos forgalmazója

Nth® GRAPHICS Display kártyák és sw.ek
kinevezett forgalmazója

ACE Térképészeti célra is alkalmas A1
méretű síkplotterek forgalmazója

AGF Tel./fax: (22) 321-359

